Грокаем алгоритмы

Иллюстрированное пособие для программистов и любопытствующих

Адитья Бхаргава



«Грокнуть» означает понять так полно, что наблюдатель становится частью объекта наблюдения...

Р. Хайнлайн

ПИТЕР


ПИТЕР®

Aditya Bhargava

Grokking Algorithms

An illustrated guide for programmers and other curious people



Адитья Бхаргава

Грокаем алгоритмы

Иллюстрированное пособие для программистов и любопытствующих



Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск

2017

Адитья Бхаргава

Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих

Серия «Библиотека программиста»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*И. Карпова*
Художник	*С. Маликова*
Корректоры	*Н. Викторова, Г. Шкатова*
Верстка	*Л. Егорова*

ББК 32.973-018
УДК 004.421

Бхаргава А.

Б94 Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. — СПб.: Питер, 2017. — 288 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02541-6

Алгоритмы — это всего лишь пошаговые алгоритмы решения задач, и большинство таких задач уже были кем-то решены, протестированы и проверены. Можно, конечно, погрузиться в глубокую философию гениального Кнута, изучить многостраничные фолианты с доказательствами и обоснованиями, но хотите ли вы тратить на это свое время? Откройте великолепно иллюстрированную книгу, и вы сразу поймете, что алгоритмы — это просто. А грокать алгоритмы — это веселое и увлекательное занятие.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1617292231 англ.
ISBN 978-5-496-02541-6

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 23.12.16. Формат 70х100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1700. Заказ № ВЗК-06096-16.

Отпечатано в АО «Первая Образцовая типография», филиал «Дом печати — ВЯТКА» в полном соответствии с качеством предоставленных материалов.
610033, г. Киров, ул. Московская, 122. Факс: (8332) 53-53-80, 62-10-36
http://www.gipp.kirov.ru; e-mail: order@gipp.kirov.ru



Оглавление

Посвящается моим родителям —
Сангите и Йогешу



Предисловие

Сначала программирование было для меня простым увлечением. Я изучил азы по книге «Visual Basic для чайников», а потом стал читать другие книги, чтобы узнать больше. Но алгоритмы мне никак не давались. Помню, как я смаковал оглавление своей первой книги по алгоритмам и думал: «Наконец-то я все узнаю!» Но материал оказался слишком сложным, и я сдался через несколько недель. Только благодаря хорошему преподавателю теории алгоритмов я понял, насколько простые и элегантные идеи заложены в ее основу.

Через несколько лет я написал свое первое иллюстрированное сообщение в блоге. Сам я визуал, поэтому мне нравится наглядный стиль изложения. С тех пор я создал немало иллюстрированных материалов по функциональному программированию, Git, машинному обучению и параллелизму. Кстати говоря, в начале своей карьеры я писал довольно посредственно. Объяснять научные концепции трудно. Чтобы придумать хорошие примеры, требуется время, чтобы объяснить сложную концепцию — тоже. Проще всего умолчать о сложных моментах. Я думал, что у меня все хорошо получается, пока после одной из моих популярных публикаций ко мне не обратился коллега со словами: «Я прочитал твой материал, но все равно ничего не понял». Мне еще предстояло многое узнать о том, как пишутся научные тексты.

В самом разгаре работы над иллюстрированными публикациями в блоге ко мне обратилось издательство *Manning* с предложением написать иллюстрированную книгу. Оказалось, что редакторы *Manning* хорошо умеют объяснять научные концепции, и они показали мне, как следует учить других. У меня была совершенно определенная цель: мне хотелось создать книгу, которая бы объясняла сложные научные темы и легко читалась. С момента написания моего первого сообщения в блоге я прошел длинный путь; надеюсь, моя книга покажется вам простой и содержательной.



Благодарности

Спасибо издательству *Manning*, которое дало мне возможность написать эту книгу и предоставило большую творческую свободу в ходе работы. Я благодарен издателю Марджану Бейсу (Marjan Bace), Майку Стивенсу (Mike Stephens) за то, что он ввел меня в курс дела, Берту Бейтсу (Bert Bates), который научил меня писать на научные темы, и Дженнифер Стаут (Jennifer Stout) — невероятно отзывчивому редактору, всегда готовому прийти на помощь. Спасибо всем участникам производственной группы Manning: Кевину Салливану (Kevin Sullivan), Мэри Пьержи (Mary Piergies), Тиффани Тейлор (Tiffany Taylor), Лесли Хаймс (Leslie Haimes) и всем остальным. Кроме того, я хочу поблагодарить всех, кто читал рукопись и делился своим мнением: Карен Бенсдон (Karen Bensdon), Роба Грина (Rob Green), Майкла Хамра (Michael Hamrah), Озрена Харловица (Ozren Harlovic), Колин Хейсти (Colin Hastie), Кристофера Хаупта (Christopher Haupt), Чака Хендерсона (Chuck Henderson), Павла Козловски (Pawel Kozlowski), Амита Ламба (Amit Lamba), Жана-Франсуа Морина (Jean-François Morin), Роберта Моррисона (Robert Morrison), Санкара Раманатана (Sankar Ramanathan), Сандера Россела (Sander Rossel), Дуга Спарлинага (Doug Sparling) и Дэмиена Уайта (Damien White).

Спасибо всем, кто помог мне в достижении цели: сотрудникам *Flashkit*, научившим меня программировать; многочисленным друзьям, которые помогали мне в работе — рецензировали главы, делились советами и предлагали разные варианты объяснений. Это были Бен Вайнгер (Ben Vinegar), Карл Пьюзон (Karl Puzon), Алекс Мэннинг (Alex Manning), Эстер Чан

(Esther Chan), Аниш Бхатт (Anish Bhatt), Майкл Гласс (Michael Glass), Никрад Махди (Nikrad Mahdi), Чарльз Ли (Charles Lee), Джаред Фридман (Jared Friedman), Хема Маникавасагам (Hema Manickavasagam), Хари Раджа (Hari Raja), Мурали Гудипати (Murali Gudipati), Шриниваса Варадан (Srinivas Varadan) и другие; также спасибо Джерри Брэди (Gerry Brady), моему учителю по теории алгоритмов. Отдельное большое спасибо таким классикам алгоритмов, как CLRS[1], Кнут и Стрэнг; безусловно, я стою на плечах гигантов.

Папа, мама, Приянка и все родные: спасибо за вашу неустанную поддержку. Огромное спасибо моей жене Мэгги. Впереди у нас много прекрасных моментов, и мне уже не придется проводить вечер пятницы за переписыванием книги.

Наконец, я хочу поблагодарить всех читателей, которые заинтересовались книгой, и тех, кто поделился своим мнением на форуме книги. Благодаря вам она действительно стала лучше.

---

[1] Авторы классической книги по алгоритмам: Кормен, Лейзерсон, Ривест, Штайн. — *Примеч. пер.*



О книге

Я прежде всего стремился к тому, чтобы книга легко читалась. Я избегаю неожиданных поворотов; каждый раз, когда в книге упоминается новая концепция, я либо объясняю ее сразу, либо говорю, где буду объяснять. Основные концепции подкрепляются упражнениями и повторными объяснениями, чтобы вы могли проверить свои предположения и убедиться в том, что не потеряли нить изложения.

В книге приводится множество примеров. Моя цель — не вывалить на читателя кучу невразумительных формул, а упростить наглядное представление этих концепций. Я также считаю, что мы лучше всего учимся тогда, когда можем вспомнить что-то уже известное, а примеры помогают освежить память. Так, когда вы вспоминаете, чем массивы отличаются от связанных списков (глава 2), просто вспомните, как ищете места для компании в кинотеатре. Наверное, вы уже поняли, что я сторонник визуального стиля обучения, — в книге полно рисунков.

Содержимое книги было тщательно продумано. Нет смысла писать книгу с описанием всех алгоритмов сортировки — для этого есть такие источники, как Википедия и *Khan Academy*. Все алгоритмы, описанные в книге, имеют практическую ценность. Я применял их в своей работе программиста, и они закладывают хорошую основу для изучения более сложных тем.

Приятного чтения!

Структура книги

В первых трех главах закладываются основы:

- **Глава 1** — вы изучите свой первый нетривиальный алгоритм: бинарный поиск. Также здесь рассматриваются основы анализа скорости алгоритмов с применением «О-большое». Эта запись часто используется в книге для описания относительной быстроты выполнения алгоритмов.

- **Глава 2** — вы познакомитесь с двумя основополагающими структурами данных: массивами и связанными списками. Эти структуры данных часто встречаются в книге и используются для создания более сложных структур данных, например хеш-таблиц (глава 5).

- **Глава 3** — вы узнаете о рекурсии — удобном приеме, используемом многими алгоритмами (например алгоритмом быстрой сортировки, о котором рассказано в главе 4).

По моему опыту, темы «О-большое» и рекурсии сложны для новичков, поэтому в этих разделах я снижаю темп изложения и привожу более подробные объяснения.

В оставшейся части книги представлены алгоритмы, часто применяемые в разных областях.

- **Методы решения задач** рассматриваются в главах 4, 8 и 9. Если вы столкнулись со сложной задачей и не знаете, как эффективно ее решить, воспользуйтесь стратегией «разделяй и властвуй» (глава 4) или методом динамического программирования (глава 9). А если вы поняли, что эффективного решения не существует, попробуйте получить приближенный ответ с использованием жадного алгоритма (глава 8).

- **Хеш-таблицы** рассматриваются в главе 5. Хеш-таблицы — исключительно полезная структура данных, предназначенная для хранения пар ключей и значений (например имени человека и адреса электронной почты или имени пользователя и пароля). Трудно переоценить практическую полезность хеш-таблиц. Приступая к решению задачи, я обычно прежде всего задаю себе два вопроса: можно ли здесь воспользоваться хеш-таблицей и можно ли смоделировать задачу в виде графа.

- **Алгоритмы графов** рассматриваются в главах 6 и 7. Графы используются для моделирования сетей: социальных, дорожных, нейронных или лю-

бых других совокупностей связей. Поиск в ширину (глава 6) и алгоритм Дейкстры (глава 7) предназначены для поиска кратчайшего расстояния между двумя точками сети: с их помощью можно вычислить кратчайший маршрут к точке назначения или количество промежуточных знакомых у двух людей в социальной сети.

- **Алгоритм *k* ближайших соседей** рассматривается в главе 10. Это простой алгоритм машинного обучения; с его помощью можно построить рекомендательную систему, механизм оптического распознавания текста, систему прогнозирования курсов акций — словом, всего, что требует прогнозирования значений («Мы думаем, что Адит поставит этому фильму 4 звезды») или классификации объектов («Это буква Q»).
- **Следующий шаг**: в главе 11 представлены 10 алгоритмов, которые хорошо подойдут для дальнейшего изучения темы.

Как работать с этой книгой

Порядок изложения и содержимое книги были тщательно продуманы. Если вас очень сильно интересует какая-то тема — переходите прямо к ней. В противном случае читайте главы по порядку, они логически переходят одна в другую.

Я настоятельно рекомендую самостоятельно выполнять код всех примеров. Вы не поверите, насколько это важно. Просто введите мои примеры кода «с листа» (или загрузите их по адресу *www.manning.com/books/grokking-algorithms* или *https://github.com/egonschiele/grokking_algorithms*) и выполните. Так у вас в памяти останется гораздо больше, чем просто при чтении.

Также я рекомендую выполнить упражнения, приведенные в книге. Упражнения не займут много времени — обычно задачи решаются за минуту или две, иногда за 5–10 минут. Упражнения помогут проверить правильность понимания материала. Если вы где-то сбились с пути, то узнаете об этом, не заходя слишком далеко.

Для кого предназначена эта книга

Эта книга предназначена для читателей, которые владеют азами программирования и хотят разобраться в алгоритмах. Может быть, вы уже столкнулись с задачей программирования и пытаетесь найти алгоритмическое решение.

А может, вы хотите понять, где вам могут пригодиться алгоритмы. Ниже приведен короткий и неполный список людей, которым может пригодиться книга:

- программисты-самоучки;
- студенты, начавшие изучать программирование;
- выпускники, желающие освежить память;
- специалисты по физике/математике/другим дисциплинам, интересующиеся программированием.

Условные обозначения и загружаемые материалы

Во всех примерах в книге используется Python 2.7. Весь программный код оформлен `моноширинным шрифтом`, чтобы его можно было отличить от обычного текста. Некоторые листинги сопровождаются аннотациями, подчеркивающими важные концепции.

Код примеров книги можно загрузить на сайте издательства по адресу *www.manning.com/books/grokking-algorithms* или *https://github.com/egonschiele/grokking_algorithms*.

Я считаю, что мы лучше всего учимся тогда, когда нам это нравится, — так что получайте удовольствие от процесса... и запускайте примеры кода!

Об авторе

Адитья Бхаргава работает программистом в Etsy, интернет-рынке авторских работ. Он получил степень магистра по информатике в Чикагском университете и ведет популярный иллюстрированный технический блог *adit.io*.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Знакомство с алгоритмами



В этой главе

- ✓ Закладываются основы для остальных глав книги.
- ✓ Вы напишете свой первый алгоритм поиска (бинарный поиск).
- ✓ Вы узнаете, как описывается время выполнения алгоритма («О-большое»).
- ✓ Будет представлен стандартный прием, часто применяемый при проектировании алгоритмов (рекурсия).

Введение

Алгоритмом называется набор инструкций для выполнения некоторой задачи. В принципе, любой фрагмент программного кода можно назвать алгоритмом, но в этой книге рассматриваются более интересные темы. Когда я отбирал алгоритмы для этой книги, я следил за тем, чтобы они были быстрыми или решали интересные задачи... или и то и другое сразу. Вот лишь несколько примеров.

- В главе 1 речь пойдет о бинарном поиске и о том, как алгоритмы могут ускорить работу кода. В одном примере алгоритм сокращает количество необходимых действий с 4 миллиардов до 32!
- Устройство GPS использует алгоритмы из теории графов (об этом в главах 6, 7 и 8) для вычисления кратчайшего пути к точке назначения.
- При помощи методов динамического программирования (см. главу 9) можно создать алгоритм для игры в шашки.

В каждом случае я опишу алгоритм и приведу пример. Затем мы обсудим время выполнения алгоритма в понятиях «О-большое». В завершение будут рассмотрены типы задач, которые могут решаться с применением того же алгоритма.

Что вы узнаете об эффективности алгоритмов

А теперь хорошая новость: скорее всего, реализация каждого алгоритма в этой книге уже доступна на вашем любимом языке программирования и вам не придется писать каждый алгоритм самостоятельно! Но любая реализация будет бесполезной, если вы не понимаете ее плюсов и минусов. В этой книге вы научитесь сравнивать сильные и слабые стороны разных алгоритмов: из каких соображений выбирать между сортировкой слиянием и быстрой сортировкой? Что использовать — массив или список? Даже выбор другой структуры данных может оказать сильное влияние на результат.

Что вы узнаете о решении задач

Вы освоите методы решения задач, которые вам сейчас, возможно, неизвестны. Примеры:

- Если вы любите создавать видеоигры, вы можете написать систему на базе искусственного интеллекта, моделирующую действия пользователя с применением алгоритмов из теории графов.
- Вы узнаете, как построить рекомендательную систему на базе k ближайших соседей.

- Некоторые проблемы не решаются за разумное время! В части книги, посвященной NP-полноте задач, рассказано о том, как идентифицировать такие задачи и построить алгоритм для получения приближенного ответа.

А если брать шире, к концу этой книги вы освоите некоторые широко применяемые алгоритмы. После этого вы сможете воспользоваться новыми знаниями для изучения более специализированных алгоритмов из области искусственного интеллекта, баз данных и т. д. или взяться за решение более сложных задач в практической работе.

ЧТО НЕОБХОДИМО ЗНАТЬ

Чтобы читать эту книгу, необходимо знать базовую алгебру. Например, возьмем следующую функцию: $f(x) = x \times 2$. Чему равен результат $f(5)$? Если вы ответили «10» — читайте спокойно.

Кроме того, вам будет проще понять эту главу (и всю книгу), если вы владеете хотя бы одним языком программирования. Все приведенные примеры написаны на Python. Если вы не знаете ни одного языка программирования, но хотите изучить — выбирайте Python: это отличный язык для начинающих. Если вы уже знаете другой язык (скажем, Ruby) — все в порядке.

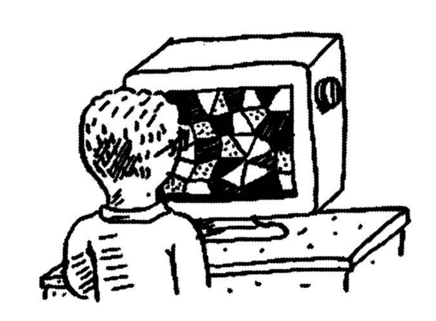
Бинарный поиск

Предположим, вы ищете фамилию человека в телефонной книге (какая древняя технология!). Она начинается с буквы «К». Конечно, можно начать с самого начала и перелистывать страницы, пока вы не доберетесь до буквы «К». Но скорее всего для ускорения поиска лучше раскрыть книгу на середине: ведь буква «К» должна находиться где-то ближе к середине телефонной книги.



Или предположим, что вы ищете слово в словаре, и оно начинается с буквы «О». И снова лучше начать с середины.

Теперь допустим, что вы вводите свои данные при входе на Facebook. При этом Facebook необходимо проверить, есть ли у вас учетная запись на сайте. Для этого ваше имя пользователя нужно найти в базе данных. Допустим, вы выбрали себе имя пользователя «karlmageddon». Facebook может начать с буквы А и проверять все подряд, но разумнее будет начать с середины.



Перед нами типичная задача поиска. И во всех этих случаях для решения задачи можно применить один алгоритм: *бинарный поиск*.

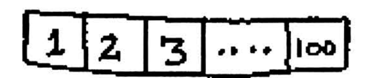
Бинарный поиск — это алгоритм; на входе он получает отсортированный список элементов (позднее я объясню, почему он должен быть отсортирован). Если элемент, который вы ищете, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден. В противном случае бинарный поиск возвращает `null`.

Например:



Ищем компанию в телефонной книге с применением бинарного поиска

Рассмотрим пример того, как работает бинарный поиск. Сыграем в простую игру: я загадал число от 1 до 100.



Вы должны отгадать мое число, использовав как можно меньше попыток. При каждой попытке я буду давать один из трех ответов: «мало», «много» или «угадал».

Предположим, вы начинаете перебирать все варианты подряд: 1, 2, 3, 4 .... Вот как это будет выглядеть.

Плохой способ угадать число

Это пример *простого поиска* (возможно, термин «*тупой поиск*» был бы уместнее). При каждой догадке исключается только одно число. Если я загадал число 99, то, чтобы добраться до него, потребуется 99 попыток!

Более эффективный поиск

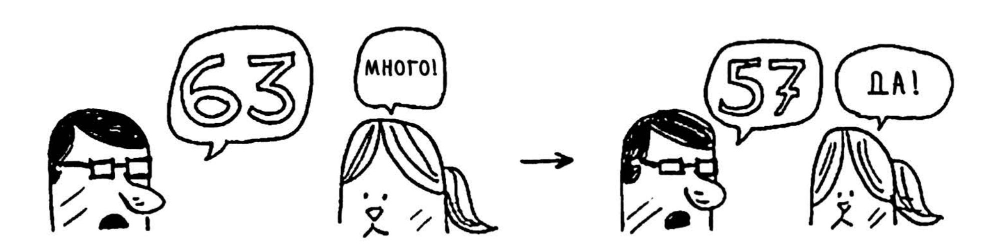
Существует другой, более эффективный способ. Начнем с 50.



Слишком мало... но вы только что исключили *половину* чисел! Теперь вы знаете, что все числа 1–50 меньше загаданного. Следующая попытка: 75.



На этот раз перелет... Но вы снова исключили половину оставшихся чисел! *С бинарным поиском вы каждый раз загадываете число в середине диапазона и исключаете половину оставшихся чисел.* Следующим будет число 63 (по середине между 50 и 75).



Так работает бинарный поиск. А вы только что узнали свой первый алгоритм! Попробуем поточнее определить, сколько чисел будет исключаться каждый раз.

При бинарном поиске каждый раз исключается половина чисел

Какое бы число я ни задумал, вы гарантированно сможете угадать его не более чем за 7 попыток, потому что с каждой попыткой исключается половина оставшихся чисел!

Предположим, вы ищете слово в словаре с 240 000 словами. Как вы думаете, сколько попыток вам понадобится в худшем случае?

ПРОСТОЙ ПОИСК: ____ ШАГОВ

БИНАРНЫЙ ПОИСК: ____ ШАГОВ

При простом поиске может потребоваться 240 000 попыток, если искомое слово находится на самой последней позиции в книге. С каждым шагом бинарного поиска количество слов сокращается вдвое, пока не останется только одно слово.

Итак, бинарный поиск потребует 18 шагов — заметная разница! В общем случае для списка из *n* элементов бинарный поиск выполняется за $\log_2 n$ шагов, тогда как простой поиск будет выполнен за *n* шагов.

ЛОГАРИФМЫ

Возможно, вы уже забыли, что такое логарифм, но наверняка помните, что такое возведение в степень. $\log_{10}100$ по сути означает, сколько раз нужно перемножить 10, чтобы получить 100. Правильный ответ — 2: 10 × 10. Итак, $\log_{10} 100 = 2$. Логарифм по смыслу противоположен возведению в степень.

$10^2 = 100 \leftrightarrow \log_{10} 100 = 2$

$10^3 = 1000 \leftrightarrow \log_{10} 1000 = 3$

$2^3 = 8 \leftrightarrow \log_2 8 = 3$

$2^4 = 16 \leftrightarrow \log_2 16 = 4$

$2^5 = 32 \leftrightarrow \log_2 32 = 5$

Логарифм — операция, обратная возведению в степень

Когда я в этой книге упоминаю «О-большое» (об этом чуть позднее), log всегда означает $\log_2$. Когда вы ищете элемент с применением простого поиска, в худшем случае вам придется проверить каждый элемент. Итак, для списка из 8 чисел понадобится не больше 8 проверок. Для бинарного поиска в худшем случае потребуется не более $\log_n$ проверок. Для списка из 8 элементов $\log_8 == 3$, потому что $2^3 == 8$. Итак, для списка из 8 чисел вам придется проверить не более 3 чисел. Для списка из 1024 элементов $\log 1024 = 10$, потому что $2^{10} == 1024$. Следовательно, для списка из 1024 чисел придется проверить не более 10 чисел.

> **ПРИМЕЧАНИЕ**
>
> Бинарный поиск работает только в том случае, если список отсортирован. Например, имена в телефонной книге хранятся в алфавитном порядке, и вы можете воспользоваться бинарным поиском. А что произойдет, если имена не будут отсортированы?

Посмотрим, как написать реализацию бинарного поиска на Python. В следующем примере кода используется массив. Если вы не знаете, как работают массивы, не беспокойтесь: эта тема рассматривается в следующей главе. Пока достаточно знать, что серию элементов можно сохранить в непрерывной последовательности ячеек, которая называется массивом. Нумерация ячеек начинается с 0: первая ячейка находится в позиции с номером 0, вторая — в позиции с номером 1 и т. д.

Функция `binary_search` получает отсортированный массив и значение. Если значение присутствует в массиве, то функция возвращает его позицию. При этом мы должны следить за тем, в какой части массива проводится поиск. Вначале это весь массив:

```
low = 0
high = len(list) - 1
```



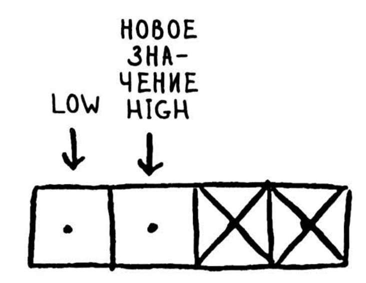
Каждый раз алгоритм проверяет средний элемент:

```
mid = (low + high) / 2
guess = list[mid]
```

Если значение (low+high) нечетно, то Python автоматически округляет значение mid в меньшую сторону

Если названное число было слишком мало, то переменная `low` обновляется соответственно:

```
if guess < item:
  low = mid + 1
```



А если догадка была слишком велика, то обновляется переменная `high`. Полный код выглядит так:

```
def binary_search(list, item):
  low = 0///
  high = len(list)-1

  while low <= high:
    mid = (low + high)
    guess = list[mid]
    if guess == item:
      return mid
    if guess > item:
      high = mid - 1
    else:
      low = mid + 1
  return None

my_list = [1, 3, 5, 7, 9]

print binary_search(my_list, 3) # => 1
print binary_search(my_list, -1) # => None
```

В переменных low и high хранятся границы той части списка, в которой выполняется поиск

Пока эта часть не сократится до одного элемента ...

... проверяем средний элемент

Значение найдено

Много

Мало

Значение не существует

А теперь протестируем функцию!

Вспомните: нумерация элементов начинается с 0. Второй ячейке соответствует индекс 1

"None" в Python означает "ничто". Это признак того, что элемент не найден

Упражнения

1.1 Имеется отсортированный список из 128 имен, и вы ищете в нем значение методом бинарного поиска. Какое максимальное количество проверок для этого может потребоваться?

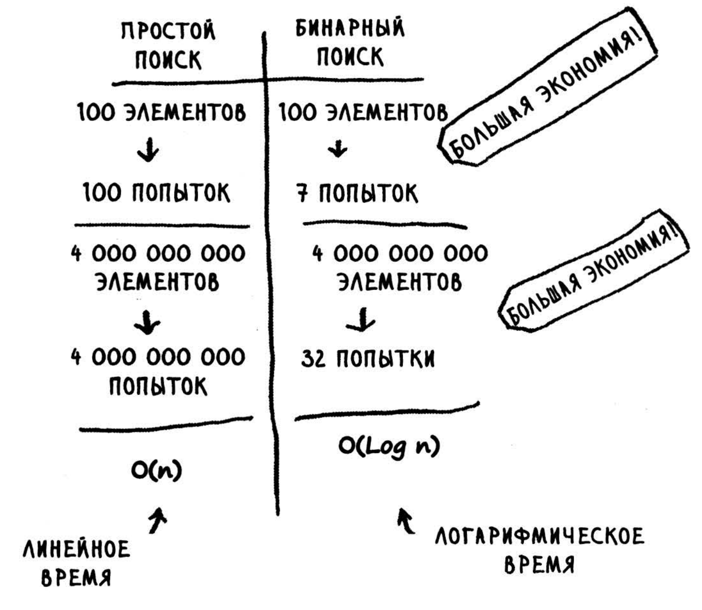
1.2 Предположим, размер списка увеличился вдвое. Как изменится максимальное количество проверок?

Время выполнения

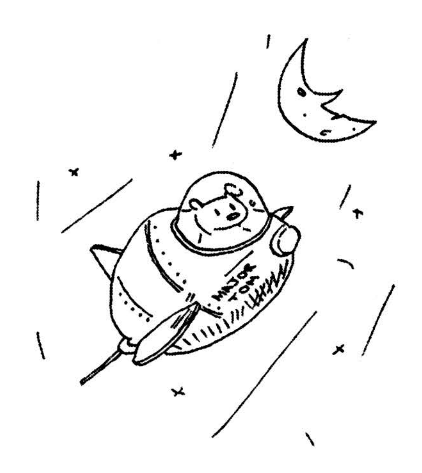
Каждый раз, когда мы будем рассматривать очередной алгоритм, я буду обсуждать время его выполнения. Обычно следует выбирать самый эффективный алгоритм, будь то оптимизация по времени или памяти.

Вернемся к бинарному поиску. Сколько времени сэкономит его применение? В первом варианте мы последовательно проверяли каждое число, одно за другим. Если список состоит из 100 чисел, может потребоваться до 100 попыток. Для списка из 4 миллиардов чисел потребуется до 4 миллиардов попыток. Таким образом, максимальное количество попыток совпадает с размером списка. Такое время выполнения называется *линейным*.

С бинарным поиском дело обстоит иначе. Если список состоит из 100 элементов, потребуется не более 7 попыток. Для списка из 4 миллиардов элементов потребуется не более 32 попыток. Впечатляет, верно? Бинарный поиск выполняется за *логарифмическое время*. В следующей таблице приводится краткая сводка результатов.



Время выполнения алгоритмов поиска



«О-большое»

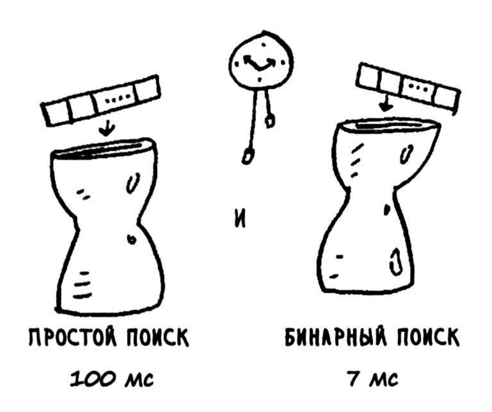
Специальная нотация *«О-большое»* описывает скорость работы алгоритма. Зачем вам это? Время от времени вам придется использовать чужие алгоритмы, а потому неплохо было бы понимать, насколько быстро или медленно они работают. В этом разделе я объясню, что представляет собой «О-большое», и приведу список самых распространенных вариантов времени выполнения для некоторых алгоритмов.

Время выполнения алгоритмов растет с разной скоростью

Боб пишет алгоритм поиска для NASA. Его алгоритм заработает, когда ракета будет подлетать к Луне, и поможет вычислить точку посадки.

Это один из примеров того, как время выполнения двух алгоритмов растет с разной скоростью. Боб пытается выбрать между простым и бинарным поиском. Его алгоритм должен работать быстро и правильно. С одной стороны, бинарный поиск работает быстрее. У Боба есть всего 10 секунд, чтобы выбрать место посадки; если он не уложится в это время, то момент для посадки будет упущен. С другой стороны, простой поиск пишется проще и вероятность ошибок в нем ниже... Конечно, Боб *совершенно* не хочет допустить ошибку в коде посадки ракеты. И тогда для пущей уверенности Боб решает измерить время выполнения обоих алгоритмов для списка из 100 элементов.

Допустим, проверка одного элемента занимает 1 миллисекунду (мс). При простом поиске Бобу придется проверить 100 элементов, поэтому поиск займет 100 мс. С другой стороны, при бинарном поиске достаточно проверить всего 7 элементов ($\log_2 100$ равен приблизительно 7), а поиск займет 7 мс. Но реальный список может содержать более миллиарда элементов. Сколько времени в таком случае потребуется для выполнения простого поиска? А при бинарном поиске? Обязательно ответьте на оба вопроса, прежде чем продолжить чтение.



Время выполнения простого и бинарного поиска для списка из 100 элементов

Боб проводит бинарный поиск с 1 миллиардом элементов, и на это уходит 30 мс ($\log_2 1\,000\,000\,000$ равен приблизительно 30). «32 мс! – думает Боб. – Бинарный поиск в 15 раз быстрее простого, потому что простой поиск для 100 элементов занял 100 мс, а бинарный поиск занял 7 мс. Значит, простой поиск займет 30 × 15 = 450 мс, верно? Гораздо меньше отведенных 10 секунд». И Боб выбирает простой поиск. Верен ли его выбор?

Нет, Боб ошибается. Глубоко ошибается. Время выполнения для простого поиска с 1 миллиардом элементов составит 1 миллиард миллисекунд, а это 11 дней! Проблема в том, что время выполнения для бинарного и простого поиска *растет с разной скоростью.*

	ПРОСТОЙ ПОИСК	БИНАРНЫЙ ПОИСК
100 ЭЛЕМЕНТОВ	100 мс	7 мс
10 000 ЭЛЕМЕНТОВ	10 секунд	14 мс
1 000 000 ЭЛЕМЕНТОВ	11 дней	32 мс

Время выполнения растет с совершенно разной скоростью!

Другими словами, с увеличением количества элементов бинарный поиск занимает чуть больше времени. А простой поиск займет *гораздо* больше времени. Таким образом, с увеличением списка бинарный список внезапно начинает работать *гораздо* быстрее простого. Боб думал, что бинарный

поиск работает в 15 раз быстрее простого, но это не так. Если список состоит из 1 миллиарда элементов, бинарный поиск работает приблизительно в 33 миллиона раз быстрее. Вот почему недостаточно знать, сколько времени должен работать алгоритм, — необходимо знать, как возрастает время выполнения с ростом размера списка. Здесь-то вам и пригодится «О-большое».



«О-большое» описывает, насколько быстро работает алгоритм. Предположим, имеется список размера n. Простой поиск должен проверить каждый элемент, поэтому ему придется выполнить n операций. Время выполнения «О-большое» имеет вид $O(n)$. Постойте, но где же секунды? А их здесь нет — «О-большое» не сообщает скорость в секундах, а *позволяет сравнить количество операций*. Оно указывает, насколько быстро возрастает время выполнения алгоритма.

А теперь другой пример. Для проверки списка размером n бинарному поиску потребуется $\log n$ операций. Как будет выглядеть «О-большое»? $O(\log n)$. В общем случае «О-большое» выглядит так:

O(n)

«О-БОЛЬШОЕ» → O

(n) ← КОЛИЧЕСТВО ОПЕРАЦИЙ

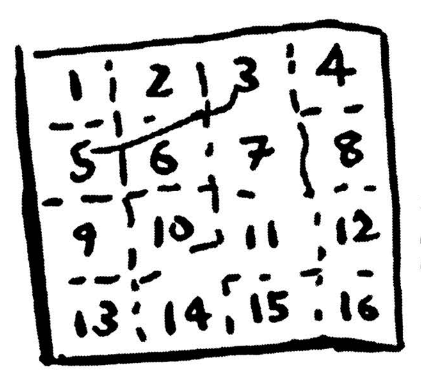
Как записывается «О-большое»

Такая запись сообщает количество операций, которые придется выполнить алгоритму. Она называется «О-большое», потому что перед количеством операций ставится символ «О» (а большое — потому что в верхнем регистре).

Теперь рассмотрим несколько примеров. Попробуйте самостоятельно оценить время выполнения этих алгоритмов.

Наглядное представление «О-большое»

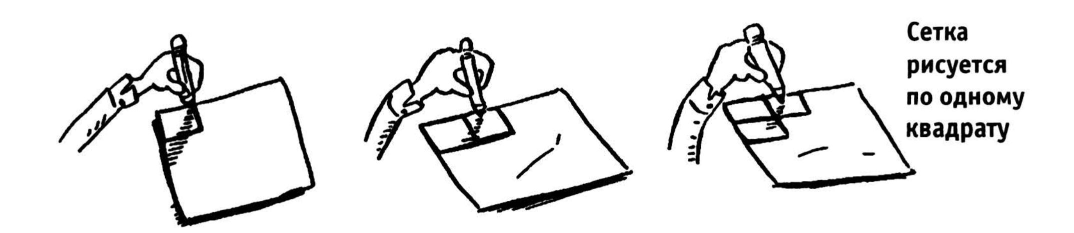
Чтобы повторить следующий практический пример, достаточно иметь несколько листков бумаги и карандаш. Допустим, вы должны построить сетку из 16 квадратов.



Как должен выглядеть хороший алгоритм для построения этой сетки?

Алгоритм 1

Как вариант можно нарисовать 16 квадратов, по одному за раз. Напоминаю: «О-большое» подсчитывает количество операций. В данном примере рисование квадрата считается одной операцией. Нужно нарисовать 16 квадратов. Сколько операций по рисованию одного квадрата придется выполнить?



Чтобы нарисовать 16 квадратов, потребуется 16 шагов. Как выглядит время выполнения этого алгоритма?

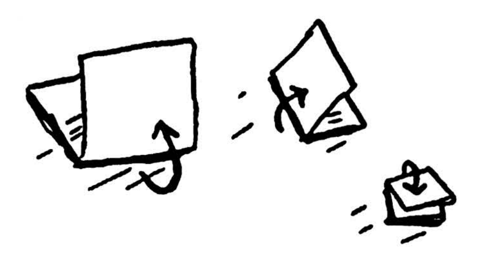
Алгоритм 2

А теперь попробуем иначе. Сложите лист пополам.

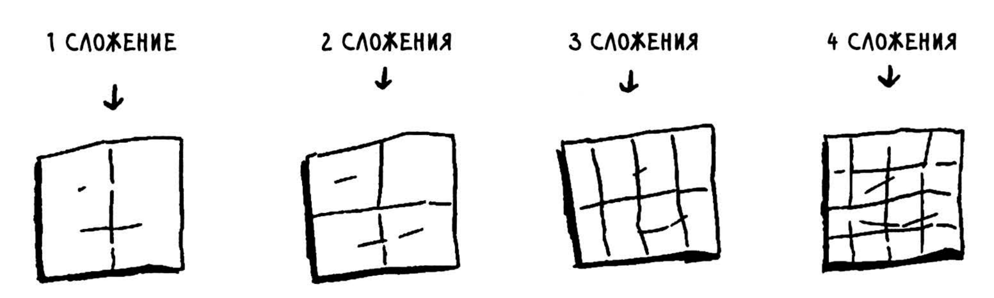


На этот раз операцией считается сложение листка. Получается, что одна операция создает сразу два прямоугольника!

Сложите бумагу еще раз, а потом еще и еще.



Разверните листок после четырех сложений — получилась замечательная сетка! Каждое сложение удваивает количество прямоугольников. За 4 операции вы создали 16 прямоугольников!



Построение сетки за 4 сложения

При каждом складывании количество прямоугольников увеличивается вдвое, так что 16 прямоугольников строятся за 4 шага. Как записать время выполнения этого алгоритма? Напишите время выполнения обоих алгоритмов, прежде чем двигаться дальше.

Ответы: алгоритм 1 выполняется за время $O(n)$, а алгоритм 2 — за время $O(\log n)$.

«О-большое» определяет время выполнения в худшем случае

Предположим, вы используете простой поиск для поиска фамилии в телефонной книге. Вы знаете, что простой поиск выполняется за время $O(n)$, то есть в худшем случае вам придется просмотреть каждую без исключения запись в телефонной книге. Но представьте, что искомая фамилия начинается на букву «А» и этот человек стоит на самом первом месте в вашей телефонной книге. В общем, вам не пришлось просматривать все записи — вы нашли нужную фамилию с первой попытки. Отработал ли алгоритм за время $O(n)$? А может, он занял время $O(1)$, потому что результат был получен с первой попытки?

Простой поиск все равно выполняется за время $O(n)$. Просто в данном случае вы нашли нужное значение моментально; это лучший возможный случай. Однако «О-большое» описывает *худший* возможный случай. Фактически вы утверждаете, что *в худшем случае* придется просмотреть каждую запись в телефонной книге по одному разу. Это и есть время $O(n)$. И это дает определенные гарантии — вы знаете, что простой поиск никогда не будет работать медленнее $O(n)$.

ПРИМЕЧАНИЕ

Наряду с временем худшего случая также полезно учитывать среднее время выполнения. Тема худшего и среднего времени выполнения обсуждается в главе 4.

Типичные примеры «О-большого»

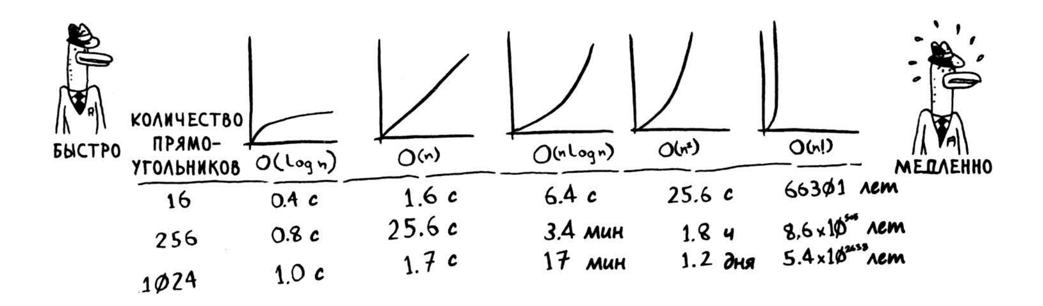
Ниже перечислены пять разновидностей «О-большого», которые будут встречаться вам особенно часто, в порядке убывания скорости выполнения:

- $O(\log n)$, или *логарифмическое время*. Пример: бинарный поиск.
- $O(n)$, или *линейное время*. Пример: простой поиск.
- $O(n * \log n)$. Пример: эффективные алгоритмы сортировки (быстрая сортировка — но об этом в главе 4).
- $O(n^2)$. Пример: медленные алгоритмы сортировки (сортировка выбором — см. главу 2).
- $O(n!)$. Пример: очень медленные алгоритмы (задача о коммивояжере — о ней будет рассказано в следующем разделе).

Предположим, вы снова строите сетку из 16 квадратов, и вы можете выбрать для решения этой задачи один из 5 алгоритмов. При использовании первого алгоритма сетка будет построена за время $O(\log n)$. В секунду выполняются до 10 операций. С временем $O(\log n)$ для построения сетки из 16 квадратов потребуются 4 операции (log 16 равен 4). Итак, сетка будет построена за 0,4 секунды. А если бы было нужно построить 1024 квадрата? На это бы потребовалось log 1024 = 10 операций, или 1 секунда. Напомню, что эти числа получены при использовании первого алгоритма.

Второй алгоритм работает медленнее: за время $O(n)$. Для построения 16 прямоугольников потребуется 16 операций, а для построения 1024 прямоугольников — 1024 операции. Сколько это составит в секундах?

Ниже показано, сколько времени потребуется для построения сетки с остальными алгоритмами, от самого быстрого до самого медленного:



Существуют и другие варианты времени выполнения, но эти пять встречаются чаще всего.

Помните, что эта запись является упрощением. На практике «О-большое» не удается легко преобразовать в количество операций с такой точностью, но пока нам хватит и этого. Мы еще вернемся к «О-большому» в главе 4, после рассмотрения еще нескольких алгоритмов. А пока перечислим основные результаты:

- Скорость алгоритмов измеряется не в секундах, а в темпе роста количества операций.
- По сути формула описывает, насколько быстро возрастает время выполнения алгоритма с увеличением размера входных данных.
- Время выполнения алгоритмов выражается как «О-большое».
- Время выполнения $O(\log n)$ быстрее $O(n)$, а с увеличением размера списка, в котором ищется значение, оно становится *намного* быстрее.

Упражнения

Приведите время выполнения «О-большое» для каждого из следующих сценариев.

1.3 Известна фамилия, нужно найти номер в телефонной книге.

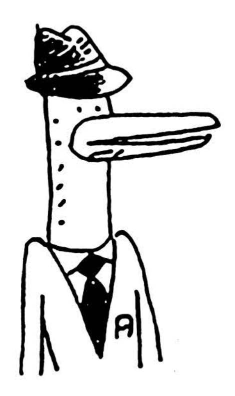
1.4 Известен номер, нужно найти фамилию в телефонной книге. (Подсказка: вам придется провести поиск по всей книге!)

1.5 Нужно прочитать телефоны всех людей в телефонной книге.

1.6 Нужно прочитать телефоны всех людей, фамилии которых начинаются с буквы «А». (Вопрос с подвохом! В нем задействованы концепции, которые более подробно рассматриваются в главе 4. Прочитайте ответ — скорее всего, он вас удивит!)

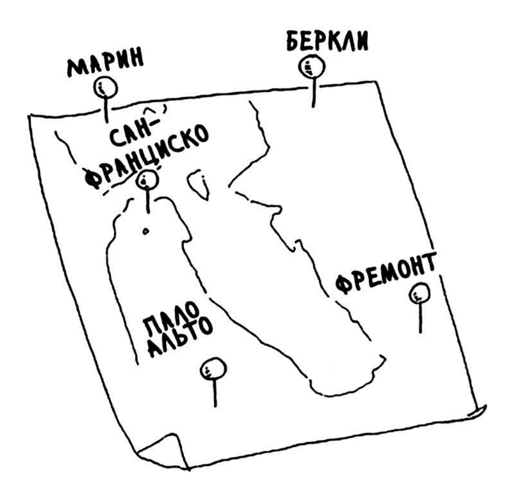
Задача о коммивояжере

Наверное, после прочтения предыдущего раздела вы подумали: «Уж мне-то точно не попадется алгоритм с временем $O(n!)$» Ошибаетесь, и я это сейчас докажу! Мы рассмотрим алгоритм с очень, очень плохим временем выполнения. Это известная задача из области теории вычислений, в которой время выполнения растет с просто ужасающей скоростью, и некоторые очень умные люди считают, что с этим ничего не поделать. Она называется *задачей о коммивояжере*.

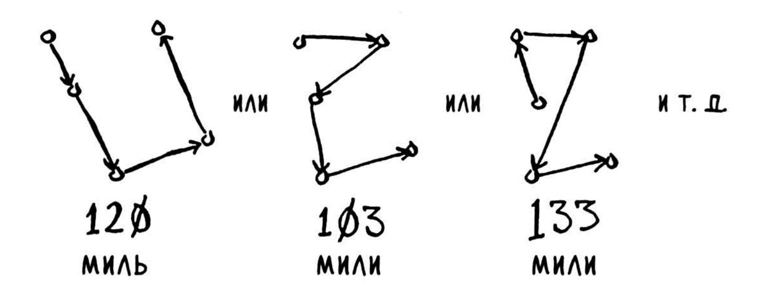


Это коммивояжер.

Он должен объехать 5 городов.



Коммивояжер хочет побывать в каждом из 5 городов так, чтобы при этом проехать минимальное общее расстояние. Одно из возможных решений: нужно перебрать все возможные комбинации порядка объезда городов.



Все расстояния суммируются, после чего выбирается путь с кратчайшим расстоянием. Для 5 городов можно создать 120 перестановок, поэтому решение задачи для 5 городов потребует 120 операций. Для 6 городов количество операций увеличивается до 720 (существуют 720 возможных перестановок). А для 7 городов потребуется уже 5040 операций!

ГОРОДА	ОПЕРАЦИИ
6	720
7	5040
8	40320
...	...
15	1,307,674,368,000
...	...
30	265,252,859,812,191,058,636,308,480,000,000

Количество операций стремительно растет

В общем случае для вычисления результата при n элементах потребуется $n!$ (n-факториал) операций. А значит, время выполнения составит $O(n!)$ (такое время называется ***факториальным***). При любом сколько-нибудь серьезном размере списка количество операций будет просто огромным. Скажем, если вы попытаетесь решить задачу для 100+ городов, сделать это вовремя не удастся — Солнце погаснет раньше.

Какой ужасный алгоритм! Значит, коммивояжер должен найти другое решение, верно? Но у него ничего не получится. Это одна из знаменитых нерешенных задач в области теории вычислений. Для нее не существует известного быстрого алгоритма, и ученые считают, что найти более эффективный алгоритм для этой задачи в принципе невозможно. В лучшем случае для нее можно поискать приближенное решение; за подробностями обращайтесь к главе 10.

И последнее замечание: если у вас уже есть опыт программирования, почитайте о бинарных деревьях поиска! Эти структуры данных кратко описаны в последней главе.

Шпаргалка

- Бинарный поиск работает намного быстрее простого.
- Время выполнения $O(\log n)$ быстрее $O(n)$, а с увеличением размера списка, в котором ищется значение, оно становится намного быстрее.
- Скорость алгоритмов не измеряется в секундах.
- Время выполнения алгоритма описывается *ростом* количества операций.
- Время выполнения алгоритмов выражается как «О-большое».

2 Сортировка выбором



В этой главе

- ✓ Вы познакомитесь с массивами и связанными списками — двумя основными структурами данных, которые используются буквально везде. Мы уже использовали массивы в главе 1 и будем использовать их почти в каждой главе книги. Массивы чрезвычайно важны, уделите им внимание! Впрочем, иногда вместо массива лучше воспользоваться связанным списком. В этой главе объясняются плюсы и минусы обеих структур данных, чтобы вы могли решить, какой вариант лучше подходит для вашего алгоритма.

- ✓ Вы изучите свой первый алгоритм сортировки. Многие алгоритмы работают только с отсортированными данными. Помните бинарный поиск? Он применяется только к предварительно отсортированному списку. В большинстве языков существуют встроенные алгоритмы сортировки, так что вам редко приходится писать свою версию «с нуля». Однако алгоритм сортировки выбором поможет перейти к алгоритму быстрой сортировки, описанному в следующей главе. Алгоритм быстрой сортировки очень важен, и вам будет проще разобраться в нем, если вы уже знаете хотя бы один алгоритм сортировки.

ЧТО НЕОБХОДИМО ЗНАТЬ

Чтобы понять ту часть этой главы, которая относится к анализу эффективности, необходимо понимать смысл понятия «О-большое» и логарифмов. Если вы совершенно не разбираетесь в этих вопросах, лучше вернуться и прочитать главу 1. «О-большое» будет использоваться в оставшихся главах книги.

Как работает память

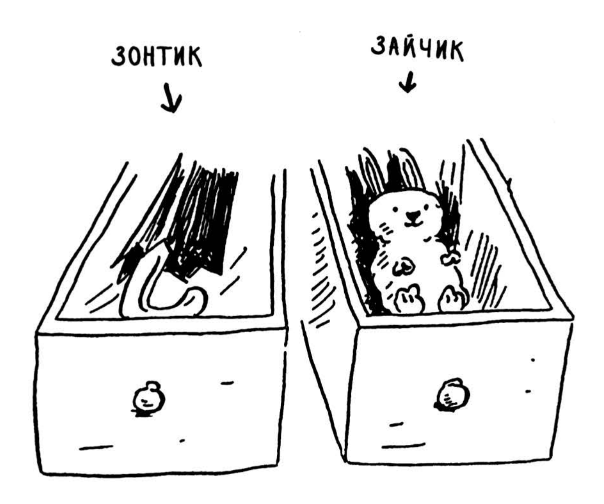
Представьте, что вы пришли в театр и хотите оставить свои личные вещи в гардеробе. Для хранения вещей есть специальные ящики.



В каждом ящике помещается один предмет. Вы хотите сдать на хранение две вещи, поэтому требуете выделить вам два ящика.

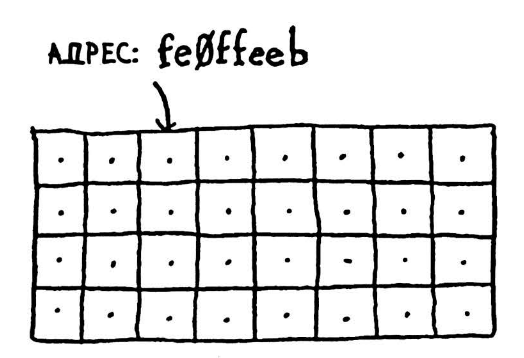


И вы оставляете свои две вещи.



Готово, можно идти на спектакль!

В сущности, именно так работает память вашего компьютера. Она представляет собой нечто вроде огромного шкафа с множеством ящиков, и у каждого ящика есть адрес.



fe0ffeeb — адрес ячейки памяти.

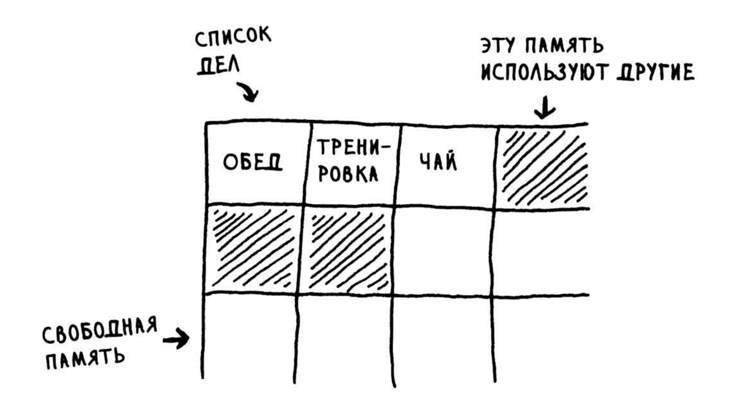
Каждый раз, когда вы хотите сохранить в памяти отдельное значение, вы запрашиваете у компьютера место в памяти, а он выдает адрес для сохранения значения. Если же вам понадобится сохранить несколько элементов, это можно сделать двумя основными способами: воспользоваться массивом или списком. В следующем разделе мы обсудим массивы и списки, их достоинства и недостатки. Не существует единственно верного способа сохранения данных на все случаи жизни, поэтому вы должны знать, чем различаются разные способы.

Массивы и связанные списки

Иногда в памяти требуется сохранить список элементов. Предположим, вы пишете приложение для управления текущими делами. Описания задач должны храниться в виде списка в памяти.



Что использовать — массив или связанный список? Для начала попробуем сохранить задачи в массиве, потому что этот способ более понятен. При использовании массива все задачи хранятся в памяти непрерывно (то есть рядом друг с другом).



Теперь предположим, что вы захотели добавить четвертую задачу. Но следующий ящик уже занят — там лежат чужие вещи!



Представьте, что вы пошли в кино с друзьями и нашли места для своей компании, но тут приходит еще один друг, и ему сесть уже некуда. Приходится искать новое место, где смогут разместиться все. В этом случае вам придется запросить у компьютера другой блок памяти, в котором поместятся все четыре задачи, а потом переместить все свои задачи туда.

Если вдруг придет еще один друг, места опять не хватит, и вам всем придется перемещаться снова! Сплошная суета. Кроме того, добавление новых элементов в массив станет серьезной проблемой. Если свободного места нет и вам каждый раз приходится перемещаться в новую область в памяти, операция добавления нового элемента будет выполняться очень медленно. Простейшее решение — «бронирование мест»: даже если список состоит всего из 3 задач, вы запрашиваете у компьютера место на 10 позиций... просто на всякий случай. Тогда в список можно будет добавить до 10 задач, и ничего перемещать не придется. Это неплохое обходное решение, но у него есть пара недостатков:

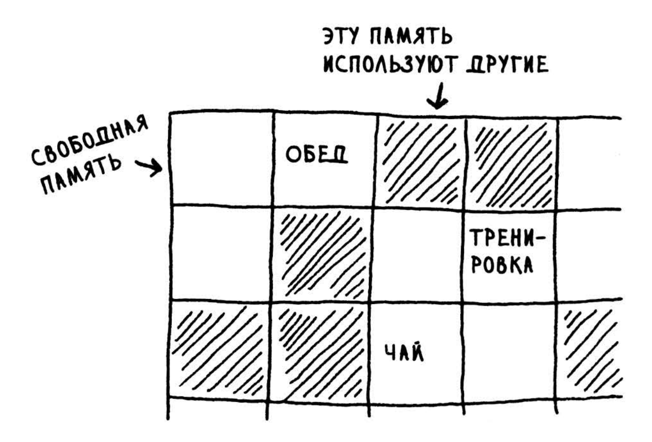
- Лишнее место может не понадобиться, и тогда память будет расходоваться неэффективно. Вы ее не используете, однако никто другой ее использовать тоже не может.

- Если в список будет добавлено более 10 задач, перемещаться все равно придется.

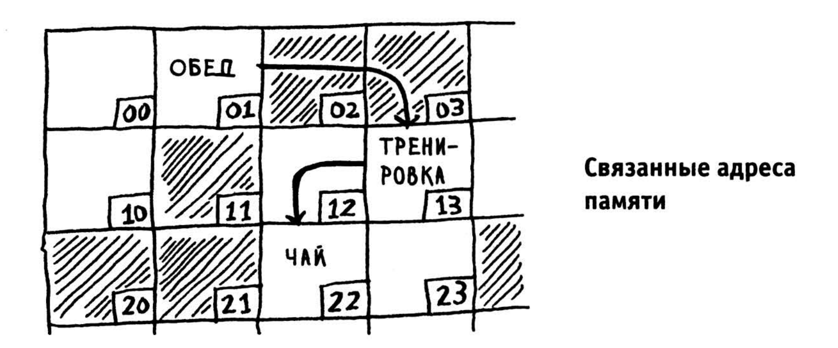
В общем, прием неплохой, но его нельзя назвать идеальным. Связанные списки решают проблему добавления новых элементов.

Связанные списки

При использовании связанного списка элементы могут размещаться где угодно в памяти.



В каждом элементе хранится адрес следующего элемента списка. Таким образом, набор произвольных адресов памяти объединяется в цепочку.



Связанные адреса памяти

Все как в игре «Найди клад». Вы приходите по первому адресу, там написано: «Следующий элемент находится по адресу 123». Вы идете по адресу 123, там написано: «Следующий элемент находится по адресу 847» и т. д. Добавить новый элемент в связанный список проще простого: просто разместите его по любому адресу памяти и сохраните этот адрес в предыдущем элементе.

Со связанными списками ничего перемещать в памяти не нужно. Также сама собой решается другая проблема: допустим, вы пришли в кино с пятью друзьями. Вы пытаетесь найти место на шестерых, но кинотеатр уже забит, и найти шесть соседних мест невозможно. Нечто похожее происходит и с массивами. Допустим, вы пытаетесь найти для массива блок на 10 000 элементов. В памяти можно найти место для 10 000 элементов, но только не смежное. Для массива не хватает места! При хранении данных в связанном списке вы фактически говорите: «Ладно, тогда садимся на свободные места и смотрим кино». Если необходимое место есть в памяти, вы сможете сохранить данные в связанном списке.

Если связанные списки так хорошо справляются со вставкой, то чем тогда хороши массивы?

Массивы

На сайтах со всевозможными хит-парадами и «первыми десятками» применяется жульническая тактика для увеличения количества просмотров. Вместо того чтобы вывести весь список на одной странице, они размещают по одному элементу на странице и заставляют вас нажимать кнопку Next для перехода к следующему элементу. Например, «Десятка лучших злодеев в сериалах» не выводится на одной странице. Вместо этого вы начинаете с № 10 (Ньюман из «Сайнфелда») и нажимаете Next на каждой странице, пока не доберетесь до № 1 (Густаво Фринг из «Во все тяжкие»). В результате сайту удается показать вам рекламу на целых 10 страницах, но нажимать Next 9 раз для перехода к первому месту скучно. Было бы гораздо лучше, если бы весь список помещался на одной странице, а вы бы могли просто щелкнуть на имени человека для получения дополнительной информации.



Похожая проблема существует и у связанных списков. Допустим, вы хотите получить последний элемент связанного списка. Просто прочитать нужное значение не удастся, потому что вы не знаете, по какому адресу оно хранится. Вместо этого придется сначала обратиться к элементу № 1 и узнать адрес элемента № 2, потом обратиться к элементу № 2 и узнать адрес элемента № 3... и так далее, пока не доберетесь до последнего элемента. Связанные списки отлично подходят в тех ситуациях, когда данные долж-

ны читаться последовательно: сначала вы читаете один элемент, по адресу переходите к следующему элементу и т. д. Но если вы намерены прыгать по списку туда-сюда, держитесь подальше от связанных списков.

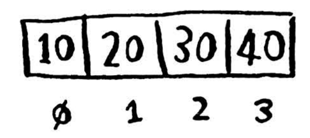
С массивами дело обстоит совершенно иначе. Работая с массивом, вы заранее знаете адрес каждого его элемента. Допустим, массив содержит пять элементов и вы знаете, что он начинается с адреса 00. По какому адресу хранится пятый элемент?



Простейшая математика дает ответ: это адрес 04. Массивы прекрасно подходят для чтения элементов в произвольных позициях, потому что обращение к любому элементу в массиве происходит мгновенно. В связанном списке элементы не хранятся рядом друг с другом, поэтому мгновенно определить позицию *i*-го элемента в памяти невозможно — нужно обратиться к первому элементу, чтобы получить адрес второго элемента, затем обратиться ко второму элементу для получения адреса третьего — и так далее, пока вы не доберетесь до *i*-го.

Терминология

Элементы массива пронумерованы, причем нумерация начинается с 0, а не с 1. Например, в этом массиве значение 20 находится в позиции 1.



А значение 10 находится в позиции 0. Неопытных программистов этот факт обычно вводит в ступор. Тем не менее выбор нулевой начальной позиции

упрощает написание кода по работе с массивами, поэтому программисты остановились на этом варианте. Почти во всех языках программирования нумерация элементов массива начинается с 0. Вскоре вы к этому привыкнете.

Позиция элемента называется его *индексом*. Таким образом, вместо того чтобы говорить «Значение 20 находится в позиции 1», правильно сказать «Значение 20 имеет индекс 1». В этой книге термин «индекс» означает то же, что и «позиция».

Ниже приведены примеры времени выполнения основных операций с массивами и списками.

	МАССИВЫ	СПИСКИ
ЧТЕНИЕ	O(1)	O(n)
ВСТАВКА	O(n)	O(1)

O(n) = ЛИНЕЙНОЕ ВРЕМЯ
O(1) = ПОСТОЯННОЕ ВРЕМЯ

Вопрос: почему вставка элемента в массив требует времени $O(n)$? Предположим, вы хотите вставить элемент в начало массива. Как бы вы это сделали? Сколько времени на это потребуется? Ответы на эти вопросы вы найдете в следующем разделе!

Упражнения

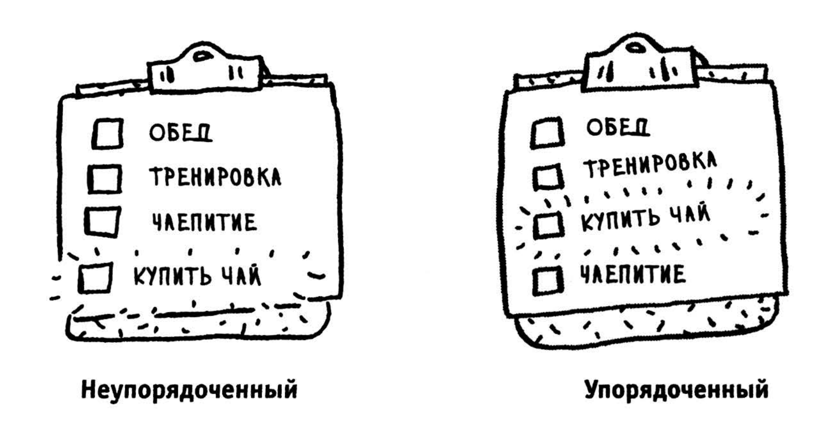
2.1 Допустим, вы строите приложение для управления финансами.

1. ПРОДУКТЫ
2. КИНО
3. ВЕЛОСИПЕДНЫЙ КЛУБ

Ежедневно вы записываете все свои траты. В конце месяца вы анализируете расходы и вычисляете, сколько денег было потрачено. При работе с данными выполняется множество операций вставки и относительно немного операций чтения. Какую структуру использовать — массив или список?

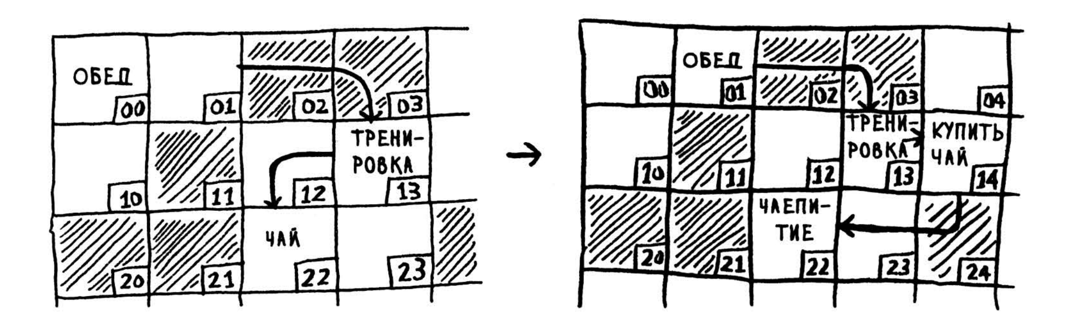
Вставка в середину списка

Предположим, вы решили, что список задач должен больше напоминать календарь. Прежде данные добавлялись только в конец списка, а теперь они должны добавляться в порядке их выполнения.



Неупорядоченный **Упорядоченный**

Что лучше подойдет для вставки элементов в середину: массивы или списки? Со списком задача решается изменением указателя в предыдущем элементе.



А при работе с массивом придется сдвигать вниз все остальные элементы.



А если свободного места не осталось, все данные придется скопировать в новую область памяти! В общем, списки лучше подходят для вставки элементов в середину.

Удаление

Что, если вы захотите удалить элемент? И снова список лучше подходит для этой операции, потому что в нем достаточно изменить указатель в предыдущем элементе. В массиве при удалении элемента все последующие элементы нужно будет сдвинуть вверх.

В отличие от вставки удаление возможно всегда. Попытка вставки может быть неудачной, если в памяти не осталось свободного места. С удалением подобных проблем не бывает.

Ниже приведены примеры времени выполнения основных операций с массивами и связанными списками.

	МАССИВЫ	СПИСКИ
ЧТЕНИЕ	$O(1)$	$O(n)$
ВСТАВКА	$O(n)$	$O(1)$
УДАЛЕНИЕ	$O(n)$	$O(1)$

Заметим, что вставка и удаление выполняются за время $O(1)$ только в том случае, если вы можете мгновенно получить доступ к удаляемому элементу. На практике обычно сохраняются ссылки на первый и последний элементы связанного списка, поэтому время удаления этих элементов составит всего $O(1)$.

Какая структура данных используется чаще: массивы или списки? Очевидно, это зависит от конкретного сценария использования. Массивы чрезвычайно популярны из-за того, что они поддерживают произвольный доступ. Всего существуют два вида доступа: *произвольный* и *последовательный*. При последовательном доступе элементы читаются по одному, начиная с первого. Связанные списки поддерживают только последовательный доступ. Если вы захотите прочитать 10-й элемент связанного списка, вам придется прочитать первые 9 элементов и перейти по ссылкам к 10-му элементу. Я часто говорю, что массивы обладают более высокой скоростью чтения; это объясняется тем, что они поддерживают произвольный доступ. Многие реальные ситуации требуют произвольного доступа, поэтому массивы часто применяются на практике. Также массивы и списки используются для реализации других структур данных (о которых будет рассказано в книге далее).

Упражнения

2.2 Допустим, вы пишете приложение для приема заказов от посетителей ресторана. Приложение должно хранить список заказов. Официанты добавляют заказы в список, а повара читают заказы из списка и выполняют их. Заказы образуют очередь: официанты добавляют заказы в конец очереди, а повар берет первый заказ из очереди и начинает готовить.

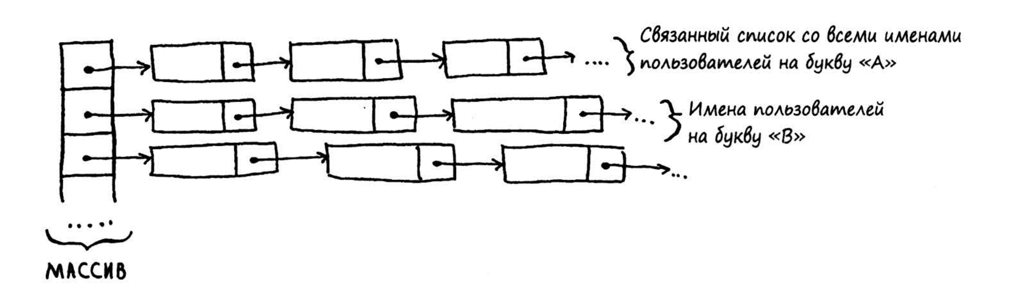


Какую структуру данных вы бы использовали для реализации этой очереди: массив или связанный список? (Подсказка: связанные списки хорошо подходят для вставки/удаления, а массивы — для произвольного доступа к элементам. Что из этого понадобится в данном случае?)

2.3 Проведем мысленный эксперимент. Допустим, Facebook хранит список имен пользователей. Когда кто-то пытается зайти на сайт Facebook, система пытается найти имя пользователя. Если имя входит в список имен зарегистрированных пользователей, то вход разрешается. Пользователи приходят на Facebook достаточно часто, поэтому поиск по списку имен пользователей будет выполняться часто. Будем считать, что Facebook использует бинарный поиск для поиска в списке. Бинарному поиску необходим произвольный доступ — алгоритм должен мгновенно обратиться к среднему элементу текущей части списка. Зная это обстоятельство, как бы вы реализовали список пользователей: в виде массива или в виде связанного списка?

2.4 Пользователи также довольно часто создают новые учетные записи на Facebook. Предположим, вы решили использовать массив для хранения списка пользователей. Какими недостатками обладает массив для выполнения вставки? Допустим, вы используете бинарный поиск для нахождения учетных данных. Что произойдет при добавлении новых пользователей в массив?

2.5 В действительности Facebook не использует ни массив, ни связанный список для хранения информации о пользователях. Рассмотрим гибридную структуру данных: массив связанных списков. Имеется массив из 26 элементов. Каждый элемент содержит ссылку на связанный список. Например, первый элемент массива указывает на связанный список всех имен пользователей, начинающихся на букву «А». Второй элемент указывает на связанный список всех имен пользователей, начинающихся на букву «В», и т. д.



Предположим, пользователь с именем «Adit B» регистрируется на Facebook и вы хотите добавить его в список. Вы обращаетесь к элементу 1 массива, находите связанный список элемента 1 и добавляете «Adit B» в конец списка. Теперь предположим, что зарегистрировать нужно пользователя «Zakhir H». Вы обращаетесь к элементу 26, который содержит связанный список всех имен, начинающихся с «Z», и проверяете, присутствует ли «Zakhir H» в этом списке.

Теперь сравните эту гибридную структуру данных с массивами и связанными списками. Будет ли она быстрее или медленнее каждой исходной структуры при поиске и вставке? Приводить «О-большое» не нужно, просто выберите одно из двух: быстрее или медленнее.

Сортировка выбором



А теперь объединим все, что вы узнали, во втором алгоритме: сортировке выбором. Чтобы освоить этот алгоритм, вы должны понимать, как работают массивы и списки и «О-большое». Допустим, у вас на компьютере записана музыка и для каждого исполнителя хранится счетчик воспроизведений.

	СЧЕТЧИК ВОС-ПРОИЗВЕДЕНИЙ
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

Вы хотите отсортировать список по убыванию счетчика воспроизведений, чтобы самые любимые исполнители стояли на первых местах. Как это сделать?

Одно из возможных решений — пройти по списку и найти исполнителя с наибольшим количеством воспроизведений. Этот исполнитель добавляется в новый список.

	СЧЕТЧИК ВОСПРОИЗВЕДЕНИЙ
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

1. У RADIOHEAD БОЛЬШЕ ВСЕГО ВОСПРОИЗВЕДЕНИЙ...

СПИСОК	СЧЕТЧИК ВОСПРОИЗВЕДЕНИЙ
RADIOHEAD	156

2. ДОБАВЛЯЕМ RADIOHEAD В НОВЫЙ СПИСОК

Потом то же самое происходит со следующим по количеству воспроизведений исполнителем.

	СЧЕТЧИК ВОСПРОИЗВЕДЕНИЙ
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

1. СЛЕДУЮЩИЙ ИСПОЛНИТЕЛЬ ПО КОЛИЧЕСТВУ ВОСПРОИЗВЕДЕНИЙ — KISHORE KUMAR

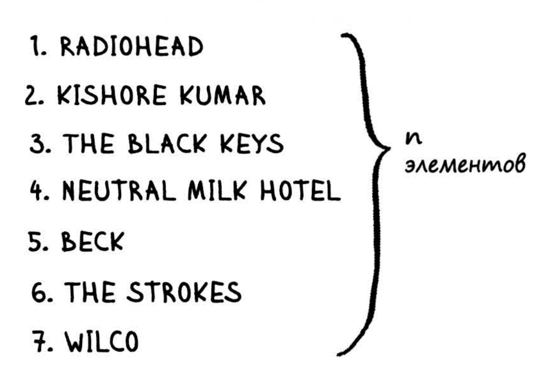
СПИСОК	СЧЕТЧИК ВОСПРОИЗВЕДЕНИЙ
RADIOHEAD	156
KISHORE KUMAR	141

2. ПОЭТОМУ ОН СЛЕДУЮЩИМ ДОБАВЛЯЕТСЯ В НОВЫЙ СПИСОК

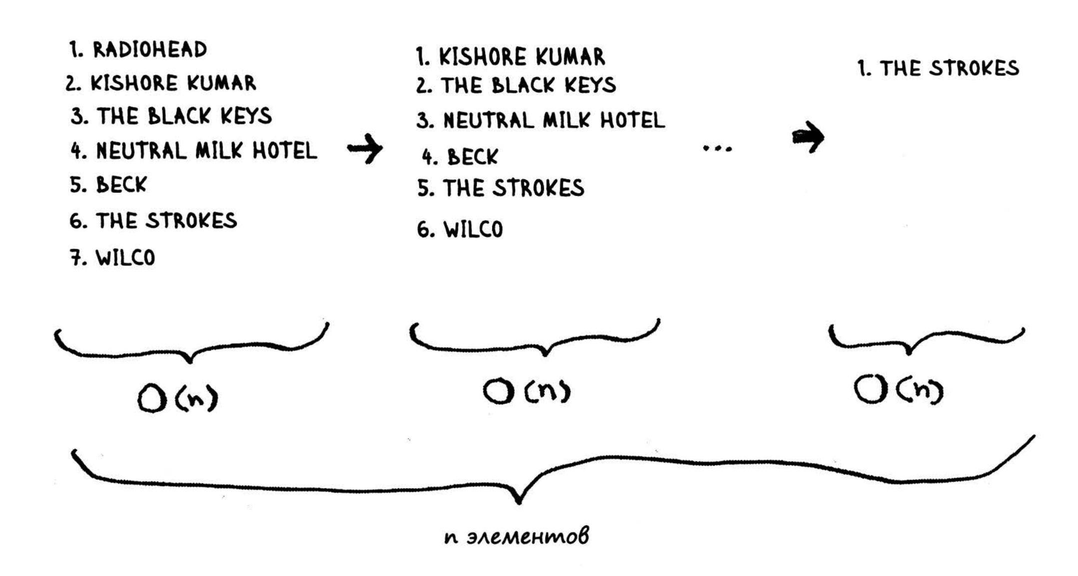
Продолжая действовать так, мы получаем отсортированный список.

	СЧЕТЧИК ВОСПРОИЗВЕДЕНИЙ
RADIOHEAD	156
KISHORE KUMAR	141
WILCO	111
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
THE BLACK KEYS	35

А теперь попробуем оценить происходящее с точки зрения теории вычислений и посмотрим, сколько времени будут занимать операции. Напомним, что время $O(n)$ означает, что вы по одному разу обращаетесь к каждому элементу списка. Например, при простом поиске по списку исполнителей каждый исполнитель будет проверен один раз.



Чтобы найти исполнителя с наибольшим значением счетчика воспроизведения, необходимо проверить каждый элемент в списке. Как вы уже видели, это делается за время $O(n)$. Итак, имеется операция, выполняемая за время $O(n)$, и ее необходимо выполнить n раз:



УМЕНЬШЕНИЕ КОЛИЧЕСТВА ПРОВЕРЯЕМЫХ ЭЛЕМЕНТОВ

Возникает закономерный вопрос: при каждом выполнении операций количество элементов, которые нужно проверить, сокращается. Со временем все сведется к проверке всего одного элемента. Почему же время выполнения все равно оценивается как $O(n^2)$? Это хороший вопрос, и ответ на него связан с ролью констант в «О-большом». Тема будет более подробно рассмотрена в главе 4, но я кратко объясню суть. Вы правы, вам действительно не нужно каждый раз проверять весь список из n элементов. Сначала проверяются n элементов, потом $n - 1$, $n - 2$ ... 2, 1. В среднем проверяется список из ½ × n элементов. Его время выполнения составит $O(n \times ½ \times n)$. Однако константы (такие как ½) в «О-большом» игнорируются (еще раз: за полным обсуждением обращайтесь к главе 4), поэтому мы просто используем $O(n \times n)$, или $O(n^2)$.

Все это требует времени $O(n \times n)$, или $O(n^2)$.

Алгоритмы сортировки очень полезны. Например, теперь вы можете отсортировать:

- имена в телефонной книге;
- даты путешествий;
- сообщения электронной почты (от новых к старым).

Алгоритм сортировки выбором легко объясняется, но медленно работает. Быстрая сортировка — эффективный алгоритм сортировки, который выполняется за время $O(n \log n)$. Но мы займемся этой темой в следующей главе!

Пример кода

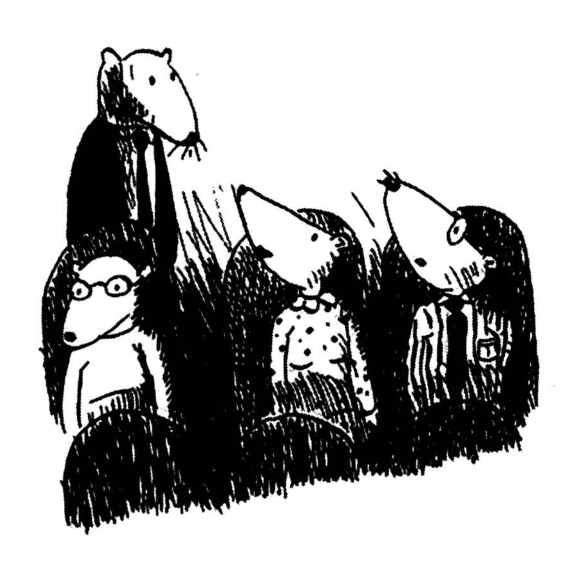
Мы не будем приводить код сортировки музыкального списка, но написанный ниже код делает нечто очень похожее: он выполняет сортировку массива по возрастанию. Напишем функцию для поиска наименьшего элемента массива:

```
def find Smallest(a r r):
  smallest = arr[0]          <······ Для хранения наименьшего значения
  smallest_index = 0         <······ Для хранения индекса наименьшего значения
  for i in range(1, len(arr)):
    if arr[i] < smallest:
      smallest = arr[i]
      smallest_index = i
  return smallest_index
```

Теперь на основе этой функции можно написать функцию сортировки выбором:

```
def selectionSort(arr):      <······ Сортирует массив
  newArr = []
  for i in range(len(arr)):
      smallest = indSmallest(arr)   <······ Находит наименьший элемент в массиве
      newArr.append(arr.pop(smallest))      и добавляет его в новый массив
  return newArr

print selectionSort([5, 3, 6, 2, 10])
```



Шпаргалка

- Память компьютера напоминает огромный шкаф с ящиками.
- Если вам потребуется сохранить набор элементов, воспользуйтесь массивом или списком.
- В массиве все элементы хранятся в памяти рядом друг с другом.
- В списке элементы распределяются в произвольных местах памяти, при этом в одном элементе хранится адрес следующего элемента.
- Массивы обеспечивают быстрое чтение.
- Списки обеспечивают быструю вставку и выполнение.
- Все элементы массива должны быть однотипными (только целые числа, только вещественные числа и т. д.).

3 Рекурсия



В этой главе

- ✓ Вы узнаете, что такое рекурсия — метод программирования, используемый во многих алгоритмах. Это важная концепция для понимания дальнейших глав книги.
- ✓ Вы научитесь разбивать задачи на базовый и рекурсивный случай. В стратегии «разделяй и властвуй» (глава 4) эта простая концепция используется для решения более сложных задач.

Эта глава мне самому очень нравится, потому что в ней рассматривается *рекурсия* — элегантный метод решения задач. Рекурсия относится к числу моих любимых тем, но вызывает у людей противоречивые чувства. Они либо обожают ее, либо ненавидят, либо ненавидят, пока не полюбят через пару-тройку лет. Лично я отношусь к третьему лагерю. Чтобы вам было проще освоить эту тему, я дам несколько советов:

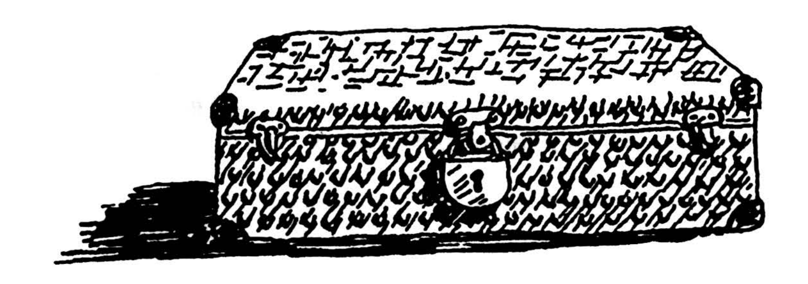
- ❑ Глава содержит множество примеров кода. Самостоятельно выполните этот код и посмотрите, как он работает.
- ❑ Мы будем рассматривать рекурсивные функции. Хотя бы один раз возьмите бумагу и карандаш и разберите, как работает рекурсивная функция: «Так, я передаю функции `factorial` значение 5, потом возвращаю

управление и передаю значение 4 функции `factorial`, которая...» и т. д. Такой разбор поможет вам понять, как работает рекурсивная функция.

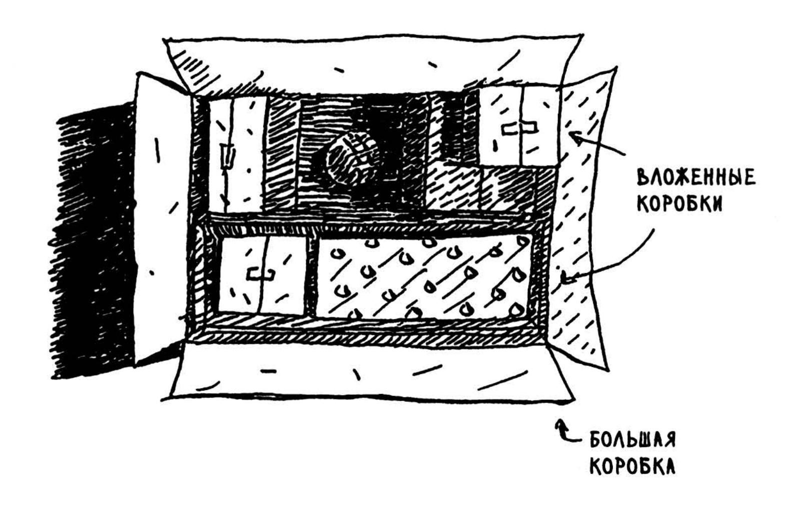
В этой главе также приводится большое количество псевдокода. *Псевдокод* представляет собой высокоуровневое описание решаемой задачи. Он записывается в форме, похожей на программный код, но в большей степени напоминает естественный язык.

Рекурсия

Допустим, вы разбираете чулан своей бабушки и натыкаетесь на загадочный запертый чемодан.

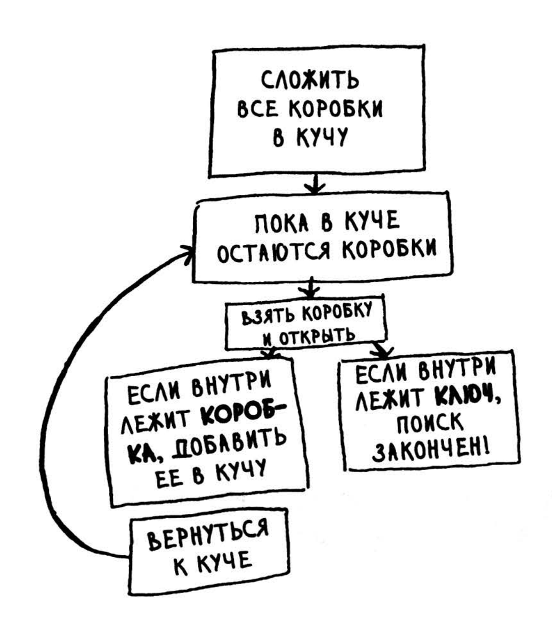


Бабушка говорит, что ключ к чемодану, скорее всего, лежит в коробке.



В коробке лежат другие коробки, а в них лежат маленькие коробочки. Ключ находится где-то там. Какой алгоритм поиска ключа предложите вы? Подумайте над алгоритмом, прежде чем продолжить чтение.

Одно из решений может выглядеть так:



1. Сложить все коробки в кучу.
2. Взять коробку и открыть.
3. Если внутри лежит коробка, добавить ее в кучу для последующего поиска.
4. Если внутри лежит ключ, поиск закончен!
5. Повторить.

Есть и альтернативное решение.



1. Просмотреть содержимое коробки.
2. Если вы найдете коробку, вернуться к шагу 1.
3. Если вы найдете ключ, поиск закончен!

Какое решение кажется вам более простым? Первое решение можно построить на цикле `while`. Пока куча коробок не пуста, взять очередную коробку и проверить ее содержимое:

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print "found the key!"
```

Второй способ основан на рекурсии. *Рекурсией* называется вызов функцией самой себя. Второе решение на псевдокоде может выглядеть так:

```
def look_for_key(b ox):
  for item in box:
    if item.is_a_box():
      look_for_key(item)       <·············  Рекурсия!
    elif item.is_a_key():
      print "found the key!"
```

Оба решения делают одно и то же, но второе решение кажется мне более понятным. Рекурсия применяется тогда, когда решение становится более понятным. Применение рекурсии не ускоряет работу программы: более того, решение с циклами иногда работает быстрее. Мне нравится одна цитата Ли Колдуэлла с сайта Stack Overlow: «Циклы могут ускорить работу программы. Рекурсия может ускорить работу программиста. Выбирайте, что важнее в вашей ситуации!»[1]

Рекурсия используется во многих нужных алгоритмах, поэтому важно понимать эту концепцию.

Базовый случай и рекурсивный случай



Так как рекурсивная функция вызывает сама себя, программисту легко ошибиться и написать функцию так, что возникнет бесконечный цикл. Предположим, вы хотите написать функцию для вывода обратного отсчета:

```
> 3...2...1
```

Ее можно записать в рекурсивном виде:

```
def countdown(i):
print i
countdow n(i-1)
```

Введите этот код и выполните его. И тут возникает проблема: эта функция выполняется бесконечно!



Бесконечный цикл

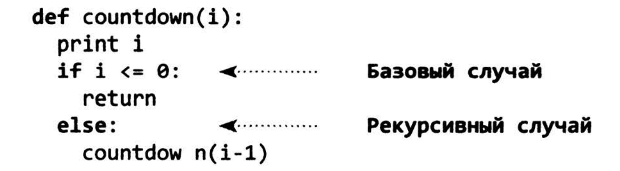
[1] http://stackoverlow.com/a/72694/139117

```
> 3...2...1...0...-1...-2...
```

Чтобы прервать выполнение сценария, нажмите Ctrl+C.

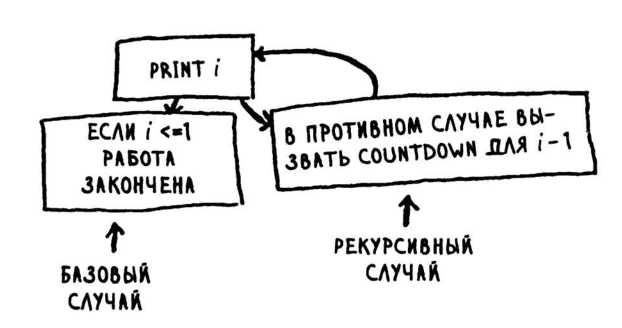
Когда вы пишете рекурсивную функцию, в ней необходимо указать, в какой момент следует прервать рекурсию. Вот почему *каждая рекурсивная функция состоит из двух частей: базового случая и рекурсивного случая.* В рекурсивном случае функция вызывает сама себя. В базовом случае функция себя не вызывает... чтобы предотвратить зацикливание.

Добавим базовый случай в функцию `countdown`:



```
def countdown(i):
  print i
  if i <= 0:        ◄·············  Базовый случай
    return
  else:             ◄·············  Рекурсивный случай
    countdow n(i-1)
```

Теперь функция работает так, как было задумано. Это выглядит примерно так:



Стек

В этом разделе рассматривается *стек вызовов*. Концепция стека вызовов играет важную роль в программировании вообще; кроме того, ее важно понимать при использовании рекурсии.

Предположим, вы устраиваете вечеринку с барбекю. Вы составляете список задач и записываете дела на листках.



Помните, когда мы рассматривали массивы и списки, у вас тоже был список задач? Задачи, то есть элементы списка, можно было добавлять и удалять в произвольных позициях списка. Стопка листков работает куда проще. Новые (вставленные) элементы добавляются в начало списка, то есть на верх стопки. Читается только верхний элемент, и он исключается из списка. Таким образом, список задач поддерживает всего два действия: *занесение* (вставка) и *извлечение* (выведение из списка и чтение.)

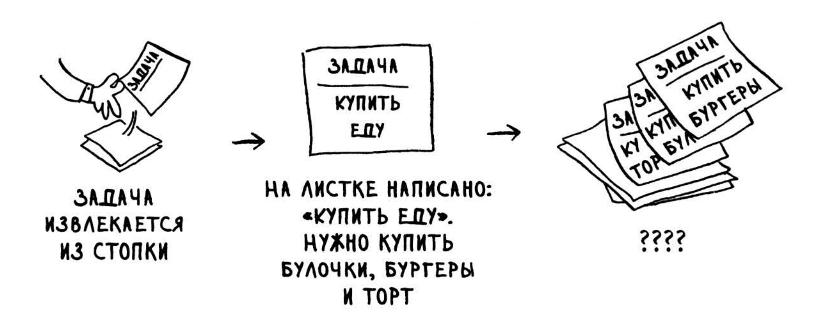


ЗАНЕСЕНИЕ
(НОВЫЙ ЭЛЕМЕНТ ДОБАВЛЯЕТСЯ НА ВЕРХ СТОПКИ)



ИЗВЛЕЧЕНИЕ
(ВЕРХНИЙ ЭЛЕМЕНТ ВЫВОДИТСЯ ИЗ СТОПКИ И ЧИТАЕТСЯ)

Посмотрим, как работает список задач:



Такая структура данных называется *стеком*. Стек — простая структура данных. А теперь самое неожиданное: все это время вы пользовались стеком, не подозревая об этом!

Стек вызовов

Во внутренней работе вашего компьютера используется стек, называемый *стеком вызовов*. Давайте посмотрим, как он работает. Предположим, имеется простая функция:

```
def greet(name):
    print "hello, " + name + "!"
    greet2(name)
    print "getting ready to say bye..."
    bye()
```

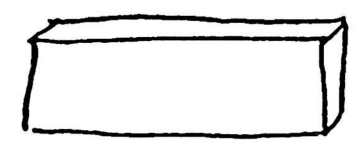
Эта функция приветствует вас, после чего вызывает две другие функции. Вот эти две функции:

```
def greet2(name):
    print "how are you, " + name + "?"
  def bye():
    print "ok bye!"
```

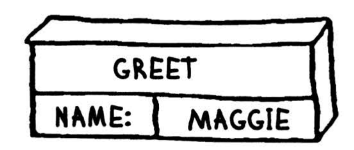
Разберемся, что происходит при вызове функции.

> **ПРИМЕЧАНИЕ**
>
> В языке Python print тоже является функцией. Чтобы не усложнять пример, мы сделаем вид, что этой функции нет. Просто подыграйте нам.

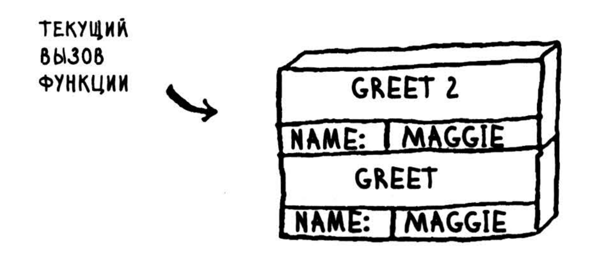
Предположим, в программе используется вызов `greet("maggie")`. Сначала ваш компьютер выделяет блок памяти для этого вызова функции.



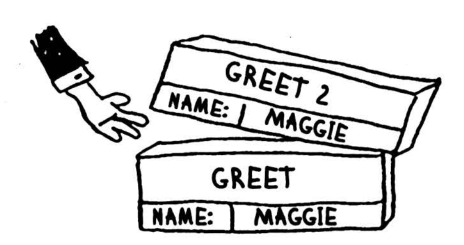
Затем эта память используется. Переменной `name` присваивается значение `"maggie"`; оно должно быть сохранено в памяти.



Каждый раз, когда вы вызываете функцию, компьютер сохраняет в памяти значения всех переменных для этого вызова. Далее выводится приветствие `hello, maggie!`, после чего следует второй вызов `greet2("maggie")`. И снова компьютер выделяет блок памяти для вызова функции.

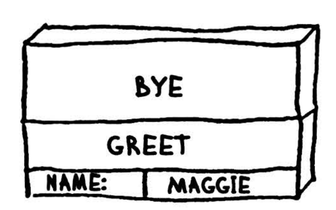


Ваш компьютер объединяет эти блоки в стек. Второй блок создается над первым. Вы выводите сообщение `how are you, maggie?`, после чего возвращаете управление из вызова функции. Когда это происходит, блок на вершине стека извлекается из него.

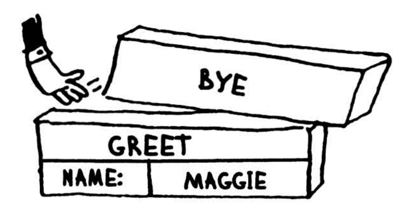


Теперь верхний блок в стеке относится к функции `greet`; это означает, что вы вернулись к функции `greet`. При вызове функции `greet2` функция `greet` еще *не была завершена*. Здесь-то и скрывается истинный смысл этого раздела: *когда вы вызываете функцию из другой функции, вызывающая функция*

приостанавливается в частично завершенном состоянии. Все значения переменных этой функции остаются в памяти. А когда выполнение функции `greet2` будет завершено, вы вернетесь к функции `greet` и продолжите ее выполнение с того места, где оно прервалось. Сначала выводится сообщение `getting ready to say bye...`, после чего вызывается функция `bye`.



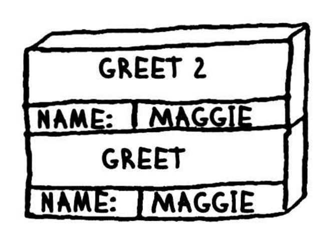
Блок для этой функции добавляется на вершину стека. Далее выводится сообщение `ok bye!` с выходом из вызова функции.



Управление снова возвращается функции `greet`. Делать больше нечего, так что управление возвращается и из функции `greet`. Этот стек, в котором сохранялись переменные разных функций, называется *стеком вызовов*.

Упражнения

3.1 Предположим, имеется стек вызовов следующего вида:



Что можно сказать о текущем состоянии программы на основании этого стека вызовов?

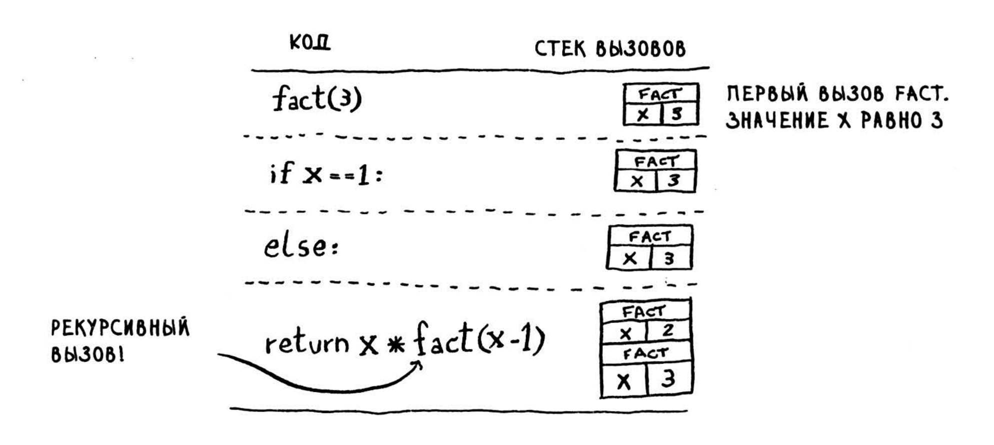
А теперь посмотрим, как работает стек вызовов с рекурсивными функциями.

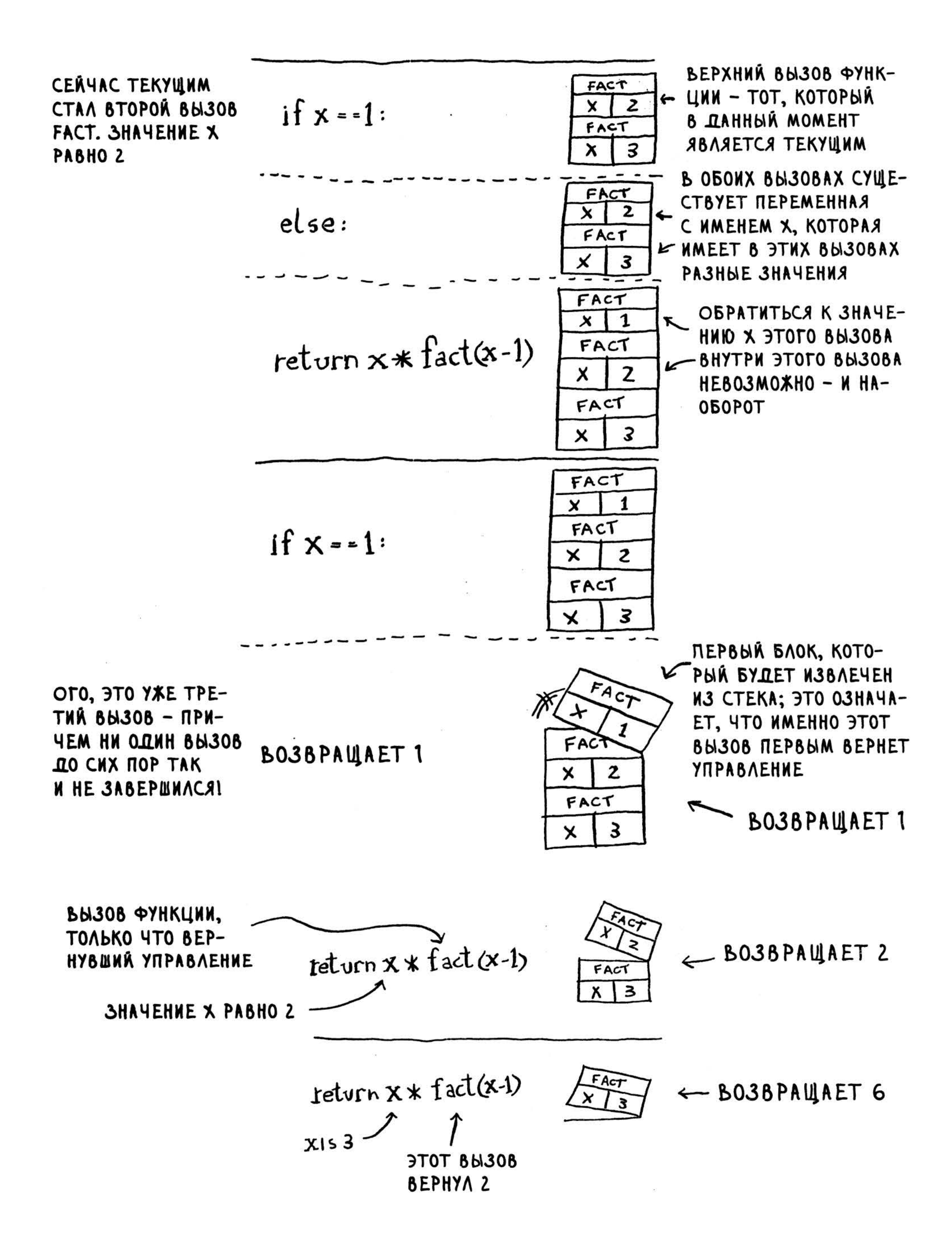
Стек вызовов с рекурсией

Рекурсивные функции тоже используют стек вызовов! Посмотрим, как это делается, на примере функции вычисления факториала. Вызов `factorial(5)` записывается в виде 5! и определяется следующим образом: 5! = 5*4*3*2*1. По тому же принципу `factorial(3)` соответствует 3*2*1. Рекурсивная функция для вычисления факториала числа выглядит так:

```
def fact(x):
  if x == 1:
    return 1
  else:
    return x * fact(x-1)
```

В программу включается вызов `fact(3)`. Проанализируем этот вызов строку за строкой и посмотрим, как изменяется стек вызовов. Стоит напомнить, что верхний блок в стеке сообщает, какой вызов `fact` является текущим.





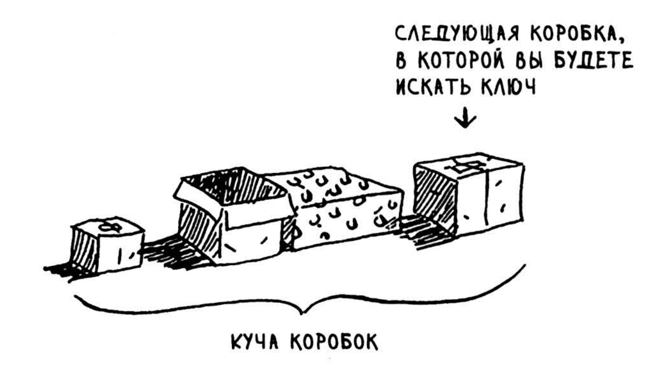
СЕЙЧАС ТЕКУЩИМ СТАЛ ВТОРОЙ ВЫЗОВ FACT. ЗНАЧЕНИЕ X РАВНО 2
if x==1:
FACT
X 2
FACT
X 3
ВЕРХНИЙ ВЫЗОВ ФУНКЦИИ – ТОТ, КОТОРЫЙ В ДАННЫЙ МОМЕНТ ЯВЛЯЕТСЯ ТЕКУЩИМ
else:
В ОБОИХ ВЫЗОВАХ СУЩЕСТВУЕТ ПЕРЕМЕННАЯ С ИМЕНЕМ X, КОТОРАЯ ИМЕЕТ В ЭТИХ ВЫЗОВАХ РАЗНЫЕ ЗНАЧЕНИЯ
return x * fact(x-1)
X 1
ОБРАТИТЬСЯ К ЗНАЧЕНИЮ X ЭТОГО ВЫЗОВА ВНУТРИ ЭТОГО ВЫЗОВА НЕВОЗМОЖНО – И НАОБОРОТ
if x==1:
ОГО, ЭТО УЖЕ ТРЕТИЙ ВЫЗОВ – ПРИЧЕМ НИ ОДИН ВЫЗОВ ДО СИХ ПОР ТАК И НЕ ЗАВЕРШИЛСЯ!
ВОЗВРАЩАЕТ 1
ПЕРВЫЙ БЛОК, КОТОРЫЙ БУДЕТ ИЗВЛЕЧЕН ИЗ СТЕКА; ЭТО ОЗНАЧАЕТ, ЧТО ИМЕННО ЭТОТ ВЫЗОВ ПЕРВЫМ ВЕРНЕТ УПРАВЛЕНИЕ
ВОЗВРАЩАЕТ 1
ВЫЗОВ ФУНКЦИИ, ТОЛЬКО ЧТО ВЕРНУВШИЙ УПРАВЛЕНИЕ
return x * fact(x-1)
ЗНАЧЕНИЕ X РАВНО 2
ВОЗВРАЩАЕТ 2
return x * fact(x-1)
ВОЗВРАЩАЕТ 6
x is 3
ЭТОТ ВЫЗОВ ВЕРНУЛ 2

Здесь важно, что каждый вызов создает собственную копию x. Обратиться к переменной x, принадлежащей другой функции, невозможно.

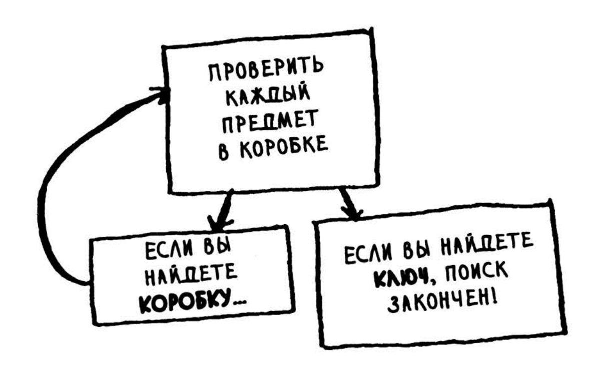
Стек играет важную роль в рекурсии. В начальном примере были представлены два решения поиска ключа. Вспомните, как выглядел первый:



В этом случае все коробки лежат в одном месте и вы всегда знаете, в каких коробках еще нужно искать ключ.



Но в рекурсивном решении никакой кучи не существует.



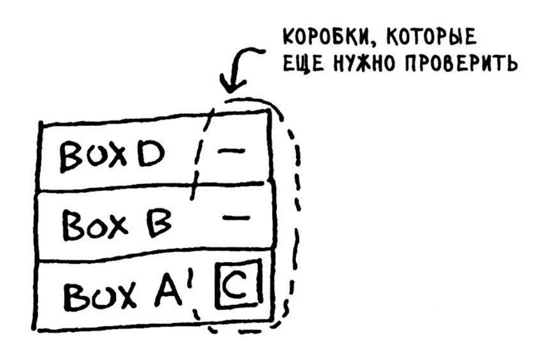
Если кучи нет, то как ваш алгоритм узнает, в каких коробках еще нужно искать? Пример:

ВЫ ПРОВЕРЯЕТЕ КОРОБКУ A → ВНУТРИ ОБНАРУЖИВАЮТСЯ КОРОБКИ B И C

ВЫ ПРОВЕРЯЕТЕ КОРОБКУ B → В НЕЙ ЛЕЖИТ КОРОБКА D

ВЫ ПРОВЕРЯЕТЕ КОРОБКУ D → ОНА ПУСТА

К этому моменту стек вызовов выглядит примерно так:



«Куча коробок» хранится в стеке! Это стек незавершенных вызовов функции, каждый из которых ведет собственный незаконченный список коробок для поиска. Стек в данном случае особенно удобен, потому что вам не нужно отслеживать коробки самостоятельно — стек делает это за вас.

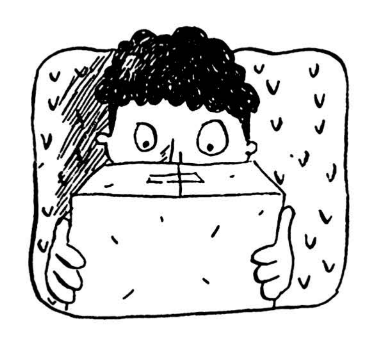
Стек удобен, но у него есть своя цена: сохранение всей промежуточной информации может привести к значительным затратам памяти. Каждый вызов функции занимает не много памяти, но если стек станет слишком высоким, это будет означать, что ваш компьютер сохраняет информацию по очень многим вызовам. На этой стадии есть два варианта:

- Переписать код с использованием цикла.
- Иногда можно воспользоваться так называемой *хвостовой рекурсией*. Это непростая тема, которая выходит за рамки книги. Вдобавок она поддерживается далеко не во всех языках.

Упражнения

3.2 Предположим, вы случайно написали рекурсивную функцию, которая бесконечно вызывает саму себя. Как вы уже видели, компьютер выделяет память в стеке при каждом вызове функции. А что произойдет со стеком при бесконечном выполнении рекурсии?

Шпаргалка



- Когда функция вызывает саму себя, это называется рекурсией.
- В каждой рекурсивной функции должно быть два случая: базовый и рекурсивный.
- Стек поддерживает две операции: занесение и извлечение элементов.
- Все вызовы функций сохраняются в стеке вызовов.
- Если стек вызовов станет очень большим, он займет слишком много памяти.

4 Быстрая сортировка



В этой главе

- ✓ Вы узнаете о стратегии «разделяй и властвуй». Случается так, что задача, над которой вы трудитесь, не решается ни одним из известных вам алгоритмов. Столкнувшись с такой задачей, хороший программист не сдается. У него существует целый арсенал приемов, которые он пытается использовать для получения решения. «Разделяй и властвуй» — первая общая стратегия, с которой вы познакомитесь.
- ✓ Далее рассматривается быстрая сортировка — элегантный алгоритм сортировки, часто применяемый на практике. Алгоритм быстрой сортировки использует стратегию «разделяй и властвуй».

Предыдущая глава была посвящена рекурсии. В этой главе вы воспользуетесь новыми знаниями для решения практических задач. Мы исследуем принцип «разделяй и властвуй», хорошо известный рекурсивный метод решения задач.

В этой главе мы постепенно добираемся до полноценных алгоритмов. В конце концов, алгоритм не особенно полезен, если он способен решать задачу только одного типа, — «разделяй и властвуй» помогает выработать новый подход к решению задач. Это всего лишь еще один инструмент в ва-

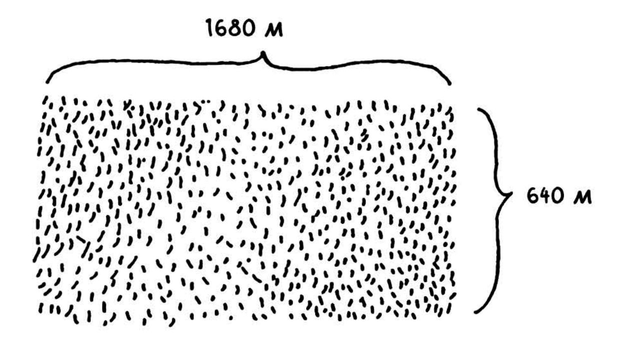
шем арсенале. Столкнувшись с новой задачей, не впадайте в ступор. Вместо этого спросите себя: «А нельзя ли решить эту задачу, применив стратегию “разделяй и властвуй”?»

К концу этой главы вы освоите свой первый серьезный алгоритм «разделяй и властвуй»: *быструю сортировку*. Этот алгоритм сортировки работает намного быстрее сортировки выбором (о которой рассказывалось в главе 2). Он является хорошим примером элегантного кода.

«Разделяй и властвуй»

Возможно, вы не сразу поймете суть стратегии «разделяй и властвуй», поэтому мы рассмотрим три примера. Сначала я приведу наглядный пример. Потом мы разберем пример кода, который выглядит не так красиво, но, пожалуй, воспринимается проще. В завершение будет рассмотрена быстрая сортировка — алгоритм сортировки, использующий стратегию «разделяй и властвуй».

Представьте, что вы фермер, владеющий земельным участком.



Вы хотите равномерно разделить землю на одинаковые *квадратные* участки. Участки должны быть настолько большими, насколько это возможно, так что ни одно из следующих решений не подойдет.



Как определить наибольший размер квадрата для участка? Воспользуйтесь стратегией «разделяй и властвуй»! Алгоритмы на базе этой стратегии являются рекурсивными.

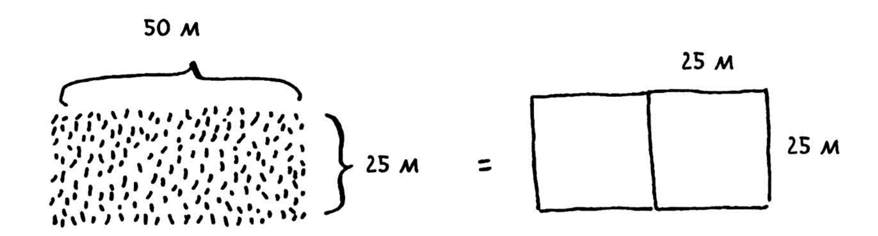
Решение задачи методом «разделяй и властвуй» состоит из двух шагов:

1. Сначала определяется базовый случай. Это должен быть простейший случай из всех возможных.

2. Задача делится или сокращается до тех пор, пока не будет сведена к базовому случаю.

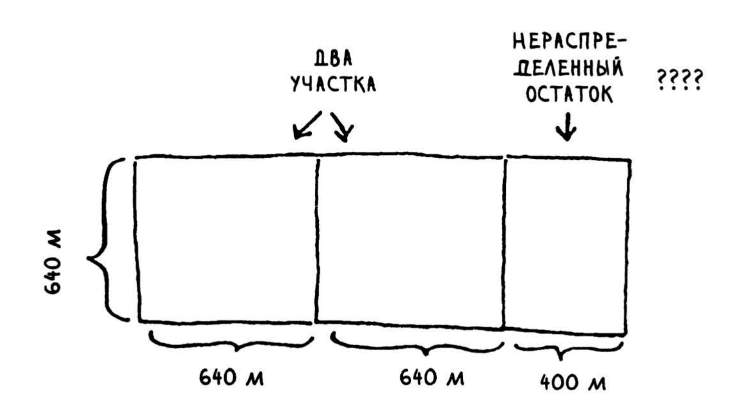
А теперь воспользуемся стратегией «разделяй и властвуй» для поиска решения этой задачи. Каков самый большой размер квадрата, который может использоваться?

Для начала нужно определить базовый случай. Самая простая ситуация — если длина одной стороны кратна длине другой стороны.

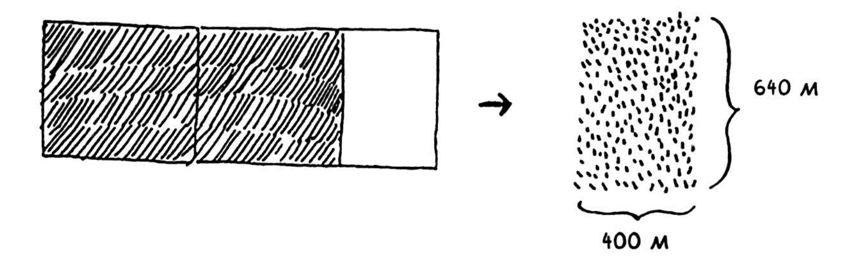


Предположим, длина одной стороны составляет 25 м, а длина другой 50 м. В этом случае размер самого большого участка составляет 25 м × 25 м, и надел после деления будет состоять из двух участков.

Теперь нужно вычислить рекурсивный случай. Здесь-то вам на помощь и приходит стратегия «разделяй и властвуй». В соответствии с ней при каждом рекурсивном вызове задача должна сокращаться. Как сократить эту задачу? Для начала разметим самые большие участки, которые можно использовать.



В исходном наделе можно разместить два участка 640 × 640, и еще останется место. Тут-то и наступает момент истины. Нераспределенный остаток — это тоже надел земли, который нужно разделить. *Так почему бы не применить к нему тот же алгоритм?*

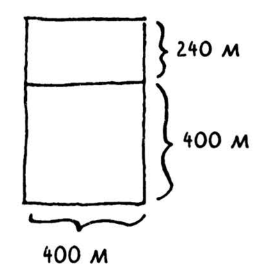


Итак, мы начали с надела 1680 × 640, который необходимо разделить на участки. Но теперь разделить нужно меньший сегмент — 640 × 400. Если

вы найдете самый большой участок, подходящий для этого размера, это будет самый большой участок, подходящий для всей фермы. Мы только что сократили задачу с размера 1680 × 640 до 640 × 400!

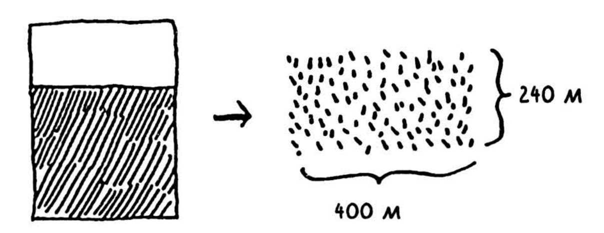
АЛГОРИТМ ЕВКЛИДА

«Если вы найдете самый большой участок, подходящий для этого размера, это будет самый большой участок, подходящий для всей фермы». Если истинность этого утверждения для вас неочевидна, не огорчайтесь. Она действительно не очевидна. К сожалению, доказательство получится слишком длинным, чтобы его можно было бы привести в книге, поэтому вам придется просто поверить мне на слово. Если вас интересует доказательство, поищите «алгоритм Евклида». Хорошее объяснение содержится на сайте Khan Academy: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm.

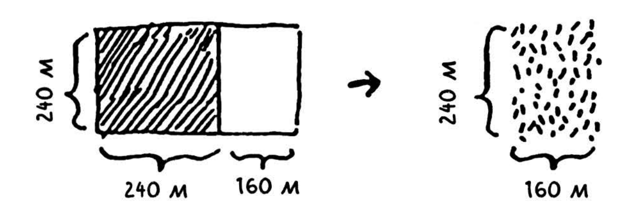


Применим тот же алгоритм снова. Если начать с участка 640 × 400, то размеры самого большого квадрата, который можно создать, составляют 400 × 400 м.

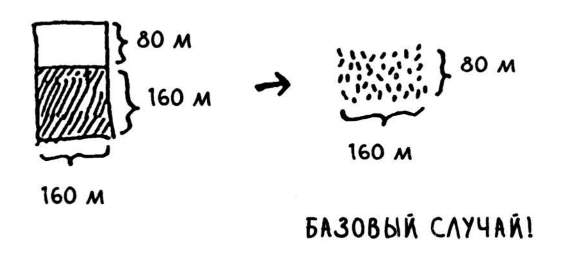
Остается меньший сегмент с размерами 400 × 240 м.



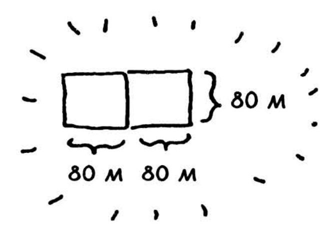
Отсекая поделенную часть, мы приходим к еще меньшему размеру сегмента, 240 × 160 м.



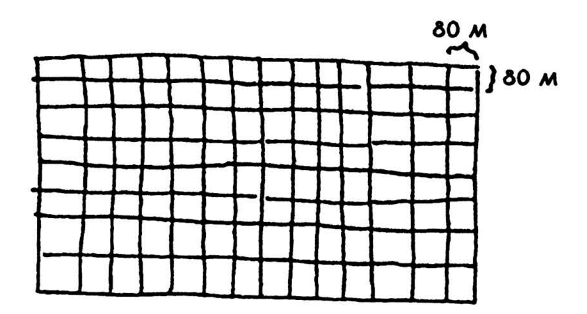
После очередного отсечения получается еще *меньший* сегмент.



Эге, да мы пришли к базовому случаю: 160 кратно 80. Если разбить этот сегмент на квадраты, ничего лишнего не останется!



Итак, для исходного надела земли самый большой размер участка будет равен 80 × 80 м.



Вспомните, как работает стратегия «разделяй и властвуй»:

1. Определите простейший случай как базовый.

2. Придумайте, как свести задачу к базовому случаю.

«Разделяй и властвуй» — не простой алгоритм, который можно применить для решения задачи. Скорее, это подход к решению задачи. Рассмотрим еще один пример.



Имеется массив чисел.

Нужно просуммировать все числа и вернуть сумму. Сделать это в цикле совсем не сложно:

```
def sum(arr):
  total = 0
  for x in arr:
    total += x
  return total

print sum([1, 2, 3, 4])
```

Но как сделать то же самое с использованием рекурсивной функции?

Шаг 1: определить базовый случай. Как выглядит самый простой массив, который вы можете получить? Подумайте, как должен выглядеть простейший случай, и продолжайте читать. Если у вас будет массив с 0 или 1 элементом, он суммируется достаточно просто.

БАЗОВЫЙ СЛУЧАЙ

[ ] 0 ЭЛЕМЕНТОВ = СУММА РАВНА 0

[7] 1 ЭЛЕМЕНТ = СУММА РАВНА 7

Итак, с базовым случаем мы определились.

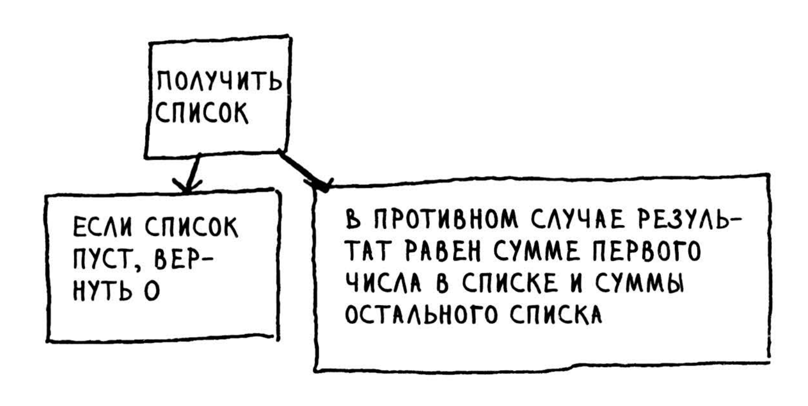
Шаг 2: каждый рекурсивный вызов должен приближать вас к пустому массиву. Как уменьшить размер задачи? Один из возможных способов:

sum([2|4|6]) = 12

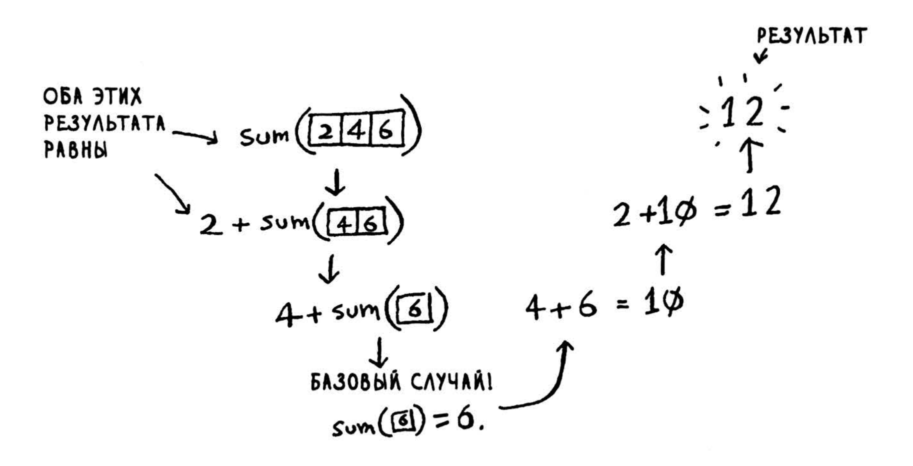
2 + sum([4|6]) = 2 + 10 = 12

В любом случае результат равен 12. Но во второй версии функции `sum` передается меньший массив. А это означает, что вы сократили размер своей задачи!

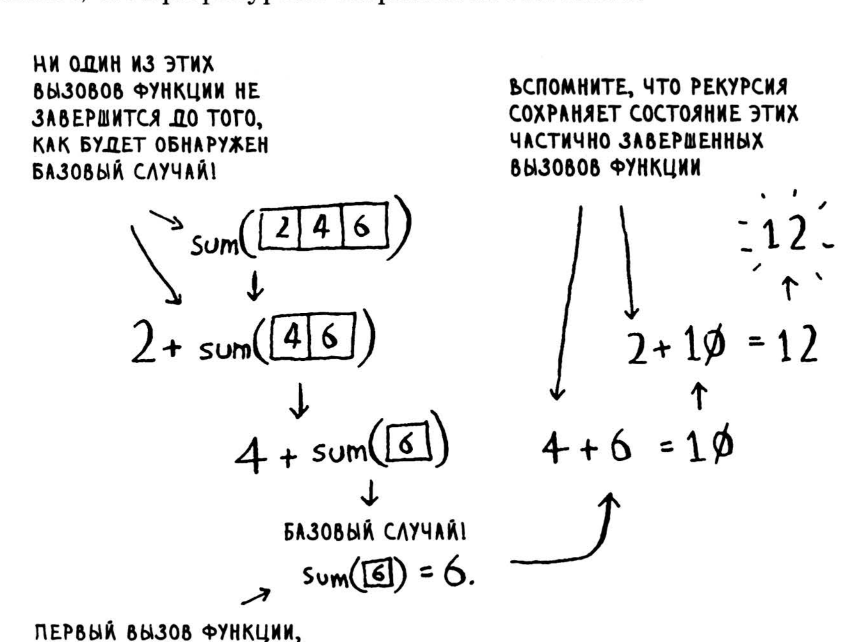
Функция `sum` может работать по следующей схеме:



А вот как это выглядит в действии.



Вспомните, что при рекурсии сохраняется состояние.



СОВЕТ

Когда вы пишете рекурсивную функцию, в которой задействован массив, базовым случаем часто оказывается пустой массив или массив из одного элемента. Если вы не знаете, с чего начать, — начните с этого.

ПАРА СЛОВ О ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ

Зачем применять рекурсию, если задача легко решается с циклом? Вполне резонный вопрос. Что ж, пора познакомиться с функциональным программированием!

В языках функционального программирования, таких как Haskell, циклов нет, поэтому для написания подобных функций приходится применять рекурсию. Если вы хорошо понимаете рекурсию, вам будет проще изучать функциональные языки. Например, вот как выглядит функция sum на языке Haskell:

```
sum [] = 0                        <.......... Базовый случай
sum (x:xs) = x + (sum xs)         <.......... Рекурсивный случай
```

На первый взгляд кажется, что одна функция имеет два определения. Первое определение выполняется для базового случая, а второе — для рекурсивного случая. Функцию также можно записать на Haskell с использованием команды if:

```
sum arr = if arr == []
            then 0
            else (head arr) + (sum (tail arr))
```

Но первое определение проще читается. Так как рекурсия широко применяется в языке Haskell, в него включены всевозможные удобства для ее использования. Если вам нравится рекурсия или вы хотите изучить новый язык — присмотритесь к Haskell.

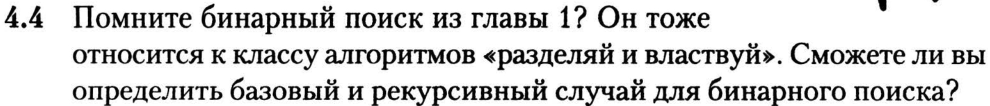
Упражнения



4.1 Напишите код для функции `sum` (см. выше).

4.2 Напишите рекурсивную функцию для подсчета элементов в списке.

4.3 Найдите наибольшее число в списке.



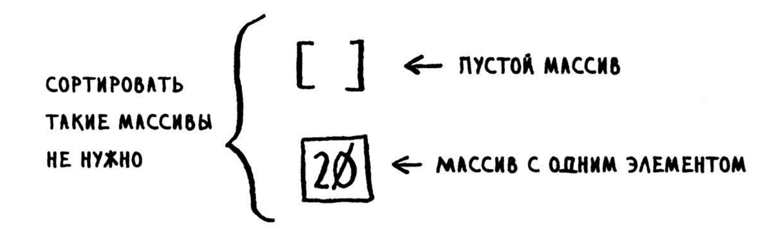
4.4 Помните бинарный поиск из главы 1? Он тоже относится к классу алгоритмов «разделяй и властвуй». Сможете ли вы определить базовый и рекурсивный случай для бинарного поиска?

Быстрая сортировка



Быстрая сортировка относится к алгоритмам сортировки. Она работает намного быстрее сортировки выбором и часто применяется в реальных программах. Например, в стандартную библиотеку C входит функция с именем `qsort`, реализующая быструю сортировку. Быстрая сортировка также основана на стратегии «разделяй и властвуй».

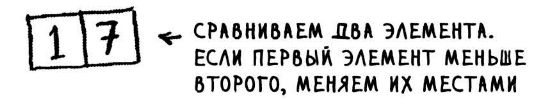
Воспользуемся быстрой сортировкой для упорядочения массива. Как выглядит самый простой массив, с которым может справиться алгоритм сортировки (помните подсказку из предыдущего раздела)? Некоторые массивы вообще не нуждаются в сортировке.



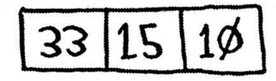
Пустые массивы и массивы, содержащие всего один элемент, станут базовым случаем. Такие массивы можно просто возвращать в исходном виде — сортировать ничего не нужно:

```
def quick sor t(array):
  if len(array) < 2:
    return array
```

Теперь перейдем к массивам большего размера. Массив из двух элементов тоже сортируется без особых проблем.



А как насчет массива из трех элементов?



Помните: мы используем стратегию «разделяй и властвуй». Следовательно, массив должен разделяться до тех пор, пока мы не придем к базовому случаю. Алгоритм быстрой сортировки работает так: сначала в массиве выбирается элемент, который называется *опорным*.



О том, как выбрать хороший опорный элемент, будет рассказано далее. А пока предположим, что опорным становится первый элемент массива.

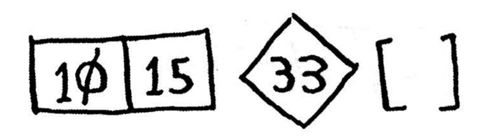
Теперь мы находим элементы, меньшие опорного, и элементы, большие опорного.



Этот процесс называется *разделением*. Теперь у вас имеются:

- подмассив всех элементов, меньших опорного;
- опорный элемент;
- подмассив всех элементов, больших опорного.

Два подмассива не отсортированы — они просто выделены из исходного массива. Но если бы они *были* отсортированы, то провести сортировку всего массива было бы несложно.

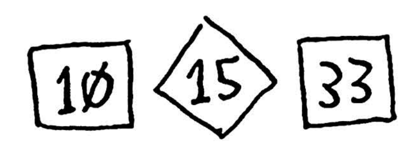


Если бы подмассивы были отсортированы, то их можно было бы объединить в порядке «левый подмассив — опорный элемент — правый подмассив» и получить отсортированный массив. В нашем примере получается `[10, 15] + [33] + [] = [10, 15, 33]`, то есть отсортированный массив.

Как отсортировать подмассивы? Базовый случай быстрой сортировки уже знает, как сортировать массивы из двух элементов (левый подмассив) и пустые массивы (правый подмассив). Следовательно, если применить алгоритм быстрой сортировки к двум подмассивам, а затем объединить результаты, получится отсортированный массив!

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]        <............ Отсортированный массив
```

Этот метод работает при любом опорном элементе. Допустим, вместо 33 в качестве опорного был выбран элемент 15.



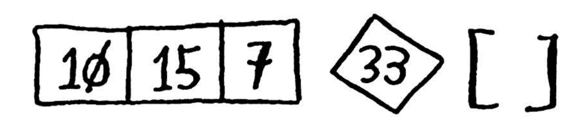
Оба подмассива состоят из одного элемента, а вы уже умеете сортировать такие подмассивы. Получается, что вы умеете сортировать массивы из трех элементов. Это делается так:

1. Выбрать опорный элемент.
2. Разделить массив на два подмассива: элементы, меньшие опорного, и элементы, большие опорного.
3. Рекурсивно применить быструю сортировку к двум подмассивам.

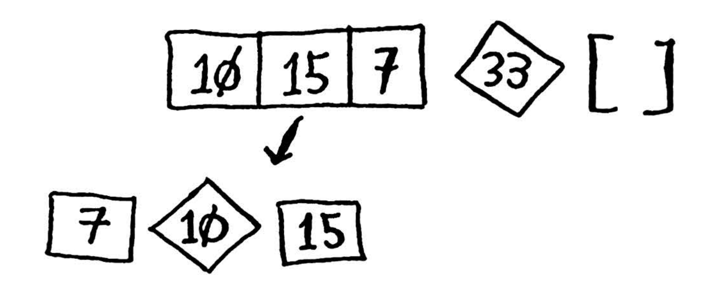
Как насчет массива из четырех элементов?



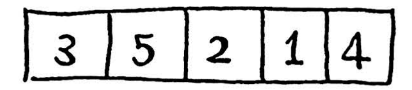
Предположим, опорным снова выбирается элемент 33.



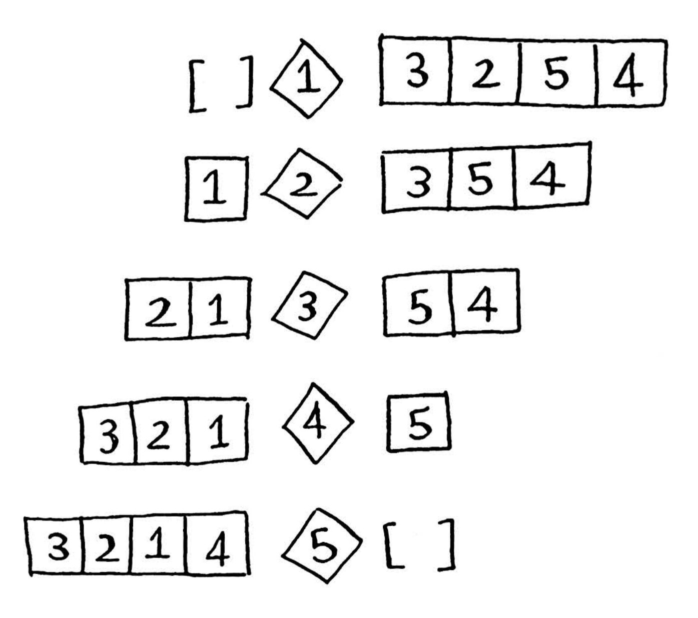
Левый подмассив состоит из трех элементов. Вы уже знаете, как сортируется массив из трех элементов: нужно рекурсивно применить к нему быструю сортировку.



Следовательно, вы можете отсортировать массив из четырех элементов. А если вы можете отсортировать массив из четырех элементов, то вы также можете отсортировать массив из пяти элементов. Почему? Допустим, имеется массив из пяти элементов.



Вот как выглядят все варианты разделения этого массива в зависимости от выбранного опорного элемента:



Все эти подмассивы содержат от 0 до 4 элементов. А вы уже знаете, как отсортировать массив, содержащий от 0 до 4 элементов, с использованием быстрой сортировки! Таким образом, независимо от выбора опорного элемента вы можете рекурсивно вызывать быструю сортировку для двух подмассивов.

Например, предположим, что в качестве опорного выбирается элемент 3. Вы применяете быструю сортировку к подмассивам.

qsort([2|1]) <3> qsort([5|4])

↓

[1|2] <3> [4|5]

↓

[1|2|3|4|5]

Подмассивы отсортированы, и теперь из них можно собрать отсортированный массив. Решение работает даже в том случае, если выбрать в качестве опорного элемент 5:

qsort([3|2|1|4]) <5> qsort([ ])

↓

[1|2|3|4] <5> [ ]

↓

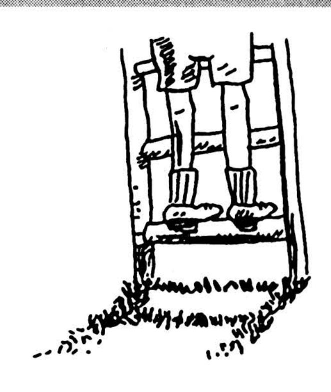
[1|2|3|4|5]

Итак, решение работает независимо от выбора опорного элемента. Следовательно, вы можете отсортировать массив из пяти элементов. По той же логике вы можете отсортировать массив из шести элементов и т. д.

ДОКАЗАТЕЛЬСТВО ПО ИНДУКЦИИ

Вы только что познакомились с методом доказательства по индукции! Это один из способов, доказывающих, что ваш алгоритм работает. Каждое индуктивное доказательство состоит из двух частей: базы (базового случая) и индукционного перехода. Звучит знакомо? Допустим, я хочу доказать, что могу подняться на самый верх стремянки. Если мои ноги стоят на ступеньке, то я могу переставить их на следующую ступеньку, — это индукционный переход. Таким образом, если я стою на ступеньке 2, то могу подняться на ступеньку 3. Что касается базового случая, я сейчас стою на ступеньке 1. Из этого следует, что я могу подняться на самый верх стремянки, каждый раз поднимаясь на одну ступеньку.

Аналогичные рассуждения применимы к быстрой сортировке. Работоспособность алгоритма для базового случая — массивов с размером 0 и 1 — была продемонстрирована. В индукционном переходе я показал, что если быстрая сортировка работает для массива из 1 элемента, то она будет работать для массива из 2 элементов. А если она работает для массивов из 2 элементов, то она будет работать для массивов из 3 элементов и т. д. Из этого можно сделать вывод, что быстрая сортировка будет работать для всех массивов любого размера. Я не буду подробно рассматривать доказательства по индукции, но это интересный метод, который идет рука об руку со стратегией «разделяй и властвуй».



А вот как выглядит программный код быстрой сортировки:

```
def quicksort(array):
  if len(array) < 2:
    return array
  else:
    pivot = array[0]
    less = [i for i in array[1:] if i <= pivot]

    greater = [i for i in array[1:] if i > pivot]

    return quicksort(less) + [pivot] + quicksort(greater)

print quicksort([10, 5, 2, 3])
```

Базовый случай: массивы с 0 и 1 элементом уже "отсортированы" (к строке `return array`)

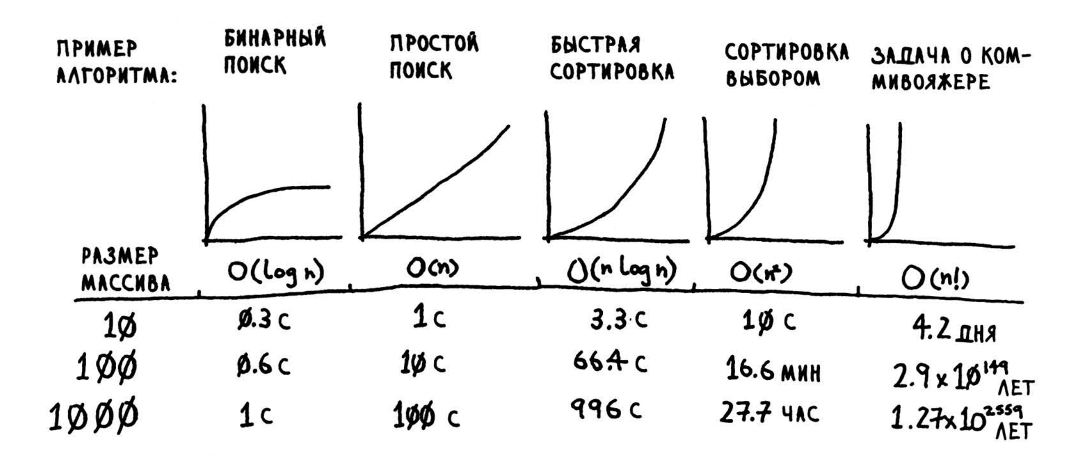
Рекурсивный случай (к строке `pivot = array[0]`)

Подмассив всех элементов, меньших опорного (к строке `less = ...`)

Подмассив всех элементов, больших опорного (к строке `greater = ...`)

Снова об «O-большом»

Алгоритм быстрой сортировки уникален тем, что его скорость зависит от выбора опорного элемента. Прежде чем рассматривать быструю сортировку, вспомним наиболее типичные варианты времени выполнения для «O-большое».



Оценки для медленного компьютера, выполняющего 10 операций в секунду

На графиках приведены примерные оценки времени при выполнении 10 операций в секунду. Они не претендуют на точность, а всего лишь дают

представление о том, насколько различается время выполнения. Конечно, на практике ваш компьютер способен выполнять гораздо больше 10 операций в секунду.

Для каждого времени выполнения также приведен пример алгоритма. Возьмем алгоритм сортировки выбором, о котором вы узнали в главе 2. Он обладает временем $O(n^2)$, и это довольно медленный алгоритм.

Другой алгоритм сортировки — так называемая *сортировка слиянием* — работает за время $O(n \log n)$. Намного быстрее! С быстрой сортировкой дело обстоит сложнее. В худшем случае быстрая сортировка работает за время $O(n^2)$.

Ничуть не лучше сортировки выбором! Но это худший случай, а в среднем быстрая сортировка выполняется за время $O(n \log n)$. Вероятно, вы спросите:

- что в данном случае понимается под *«худшим»* и *«средним»* случаем?
- если быстрая сортировка в среднем выполняется за время $O(n \log n)$, а сортировка слиянием выполняется за время $O(n \log n)$ всегда, то почему бы не использовать сортировку слиянием? Разве она не быстрее?

Сортировка слиянием и быстрая сортировка

Допустим, у вас имеется простая функция для вывода каждого элемента в списке:

```
def print_items(list):
  for item in list:
    print item
```

Эта функция последовательно перебирает все элементы списка и выводит их. Так как функция перебирает весь список, она выполняется за время $O(n)$. Теперь предположим, что вы изменили эту функцию и она делает секундную паузу перед выводом:

```
from time import sleep
def print_items2(list):
  for item in list:
    sleep(1)
    print item
```

Перед выводом элемента функция делает паузу продолжительностью в 1 секунду. Предположим, вы выводите список из пяти элементов с использованием обеих функций:

2 4 6 8 10

print_items: 2 4 6 8 10

print_items2: <ПАУЗА> 2 <ПАУЗА> 4 <ПАУЗА> 6 <ПАУЗА> 8 <ПАУЗА> 10

Обе функции проходят по списку один раз, и обе выполняются за время $O(n)$. Как вы думаете, какая из них работает быстрее? Я думаю, `print_items` работает намного быстрее, потому что она не делает паузу перед выводом каждого элемента. Следовательно, даже при том, что обе функции имеют одинаковую скорость «О-большое», реально `print_items` работает быстрее. Когда вы используете «О-большое» (например, $O(n)$), в действительности это означает следующее:

$c * n$

ФИКСИРОВАННЫЙ ПРОМЕЖУТОК ВРЕМЕНИ

Здесь `c` — некоторый фиксированный промежуток времени для вашего алгоритма. Он называется *константой*. Например, время выполнения может составлять *10 миллисекунд * n* для `print_items` против *1 секунды * n* для `print_items2`.

Обычно константа игнорируется, потому что если два алгоритма имеют разное время «О-большое», она роли не играет. Для примера возьмем бинарный и простой поиск. Допустим, такие константы присутствуют в обоих алгоритмах.

10мс * n — ПРОСТОЙ ПОИСК

1с * log n — БИНАРНЫЙ ПОИСК

Первая реакция: «Ого! У простого поиска константа равна 10 миллисекундам, а у бинарного поиска – 1 секунда. Простой поиск намного быстрее!» Теперь предположим, что поиск ведется по списку из 4 миллиардов элементов. Время будет таким:

ПРОСТОЙ ПОИСК	10мс * 4 МИЛЛИАРДА	=	463 ДНЯ
БИНАРНЫЙ ПОИСК	1с * 32	=	32 СЕКУНДЫ

Как видите, бинарный поиск все равно работает намного быстрее. Константа ни на что не повлияла.

Однако в некоторых случаях константа *может* иметь значение. Один из примеров такого рода — быстрая сортировка и сортировка слиянием. У быстрой сортировки константа меньше, чем у сортировки слиянием, поэтому, несмотря на то что оба алгоритма характеризуются временем $O(n \log n)$, быстрая сортировка работает быстрее. А на практике быстрая сортировка работает быстрее, потому что средний случай встречается намного чаще худшего.

А теперь ответим на первый вопрос: как выглядит средний случай по сравнению с худшим?

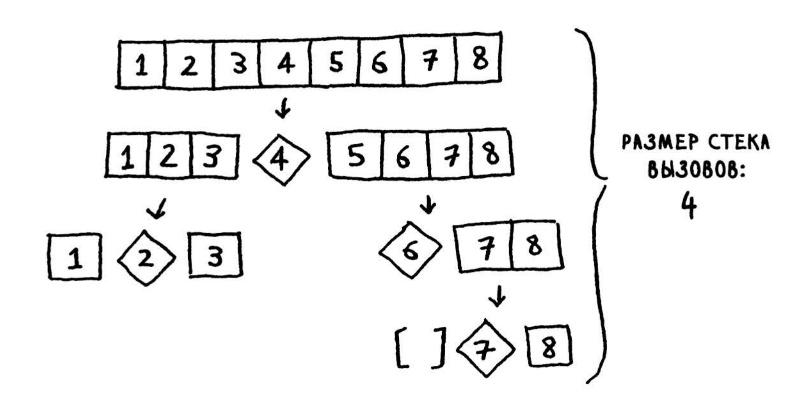
Средний и худший случай

Быстродействие быстрой сортировки сильно зависит от выбора опорного элемента. Предположим, опорным всегда выбирается первый элемент, а быстрая сортировка применяется к *уже отсортированному* массиву. Быстрая сортировка не проверяет, отсортирован входной массив или нет, и все равно пытается его отсортировать.

В КАЧЕСТВЕ ОПОРНОГО КАЖДЫЙ РАЗ ВЫБИРАЕТСЯ ПЕРВЫЙ ЭЛЕМЕНТ

ОБЩАЯ ВЫСОТА СТЕКА ВЫЗОВОВ: 8

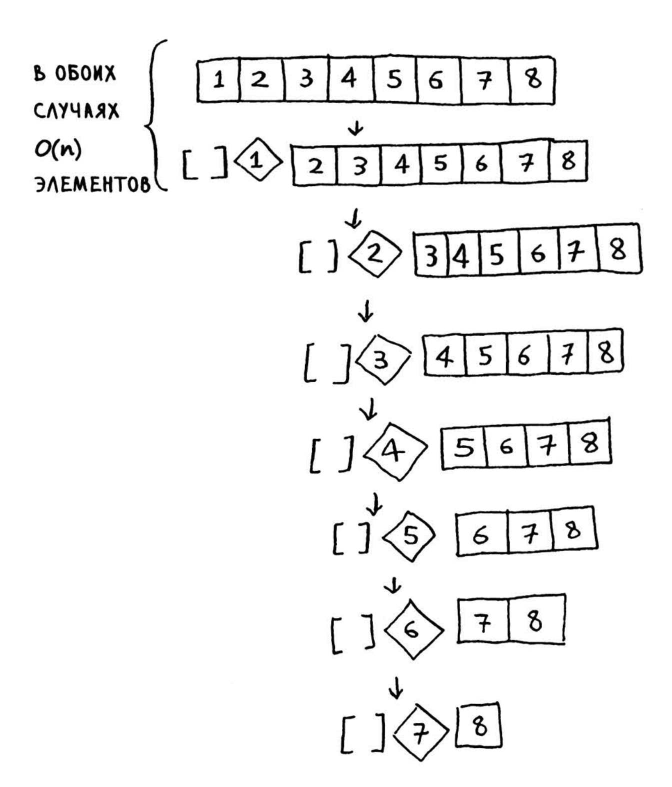
Обратите внимание: на этот раз массив не разделяется на две половины. Вместо этого один из двух подмассивов всегда пуст, так что стек вызовов получается очень длинным. Теперь предположим, что в качестве опорного всегда выбирается средний элемент. Посмотрим, как выглядит стек вызовов в этом случае.



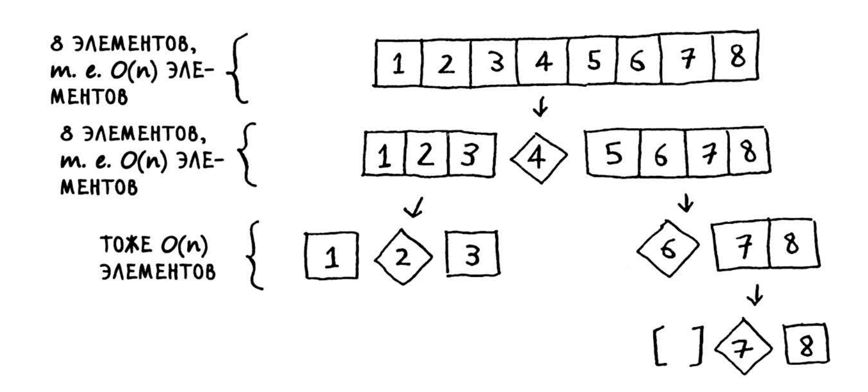
Стек намного короче! Массив каждый раз делится надвое, поэтому такое количество рекурсивных вызовов излишне. Вы быстрее добираетесь до базового случая, и стек вызовов получается более коротким.

Первый из рассмотренных примеров описывает худший сценарий, а второй — лучший. В худшем случае размер стека описывается как $O(n)$. В лучшем случае он составит $O(\log n)$.

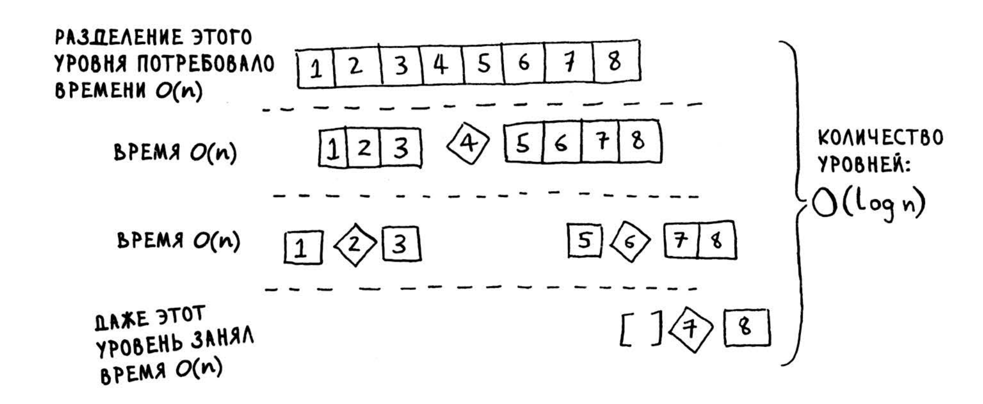
Теперь рассмотрим первый уровень стека. Один элемент выбирается опорным, а остальные элементы делятся на подмассивы. Вы перебираете все восемь элементов массива, поэтому первая операция выполняется за время $O(n)$. На этом уровне стека вызовов вы обратились ко всем восьми элементам. Но на самом деле вы обращаетесь к $O(n)$ элементам на каждом уровне стека вызовов!



Даже если массив будет разделен другим способом, вы все равно каждый раз обращаетесь к $O(n)$ элементам.



Итак, завершение каждого уровня требует времени $O(n)$.



В этом примере существуют $O(\log n)$ (с технической точки зрения правильнее сказать «высота стека вызовов равна $O(\log n)$») уровней. А так как каждый уровень занимает время $O(n)$, то весь алгоритм займет время $O(n) * O(\log n) = O(n \log n)$. Это сценарий лучшего случая.

В худшем случае существуют $O(n)$ уровней, поэтому алгоритм займет время $O(n) * O(n) = O(n^2)$.

А теперь сюрприз: лучший случай также является средним. Если вы всегда будете выбирать опорным элементом случайный элемент в массиве, быстрая сортировка в среднем завершится за время $O(n \log n)$. Это один из самых быстрых существующих алгоритмов сортировки, который заодно является хорошим примером стратегии «разделяй и властвуй».

Упражнения

Запишите «О-большое» для каждой из следующих операций ?

4.5 Вывод значения каждого элемента массива.

4.6 Удвоение значения каждого элемента массива.

4.7 Удвоение значения только первого элемента массива.

4.8 Создание таблицы умножения для всех элементов массива. Например, если массив состоит из элементов [2, 3, 7, 8, 10], сначала каждый элемент умножается на 2, затем каждый элемент умножается на 3, затем на 7 и т. д.

Шпаргалка

- Стратегия «разделяй и властвуй» основана на разбиении задачи на уменьшающиеся фрагменты. Если вы используете стратегию «разделяй и властвуй» со списком, то базовым случаем, скорее всего, является пустой массив или массив из одного элемента.
- Если вы реализуете алгоритм быстрой сортировки, выберите в качестве опорного случайный элемент. Среднее время выполнения быстрой сортировки составляет $O(n \log n)$!
- Константы в «О-большом» иногда могут иметь значение. Именно по этой причине быстрая сортировка быстрее сортировки слиянием.
- При сравнении простой сортировки с бинарной константа почти никогда роли не играет, потому что $O(\log n)$ слишком сильно превосходит $O(n)$ по скорости при большом размере списка.

5 Хеш-таблицы

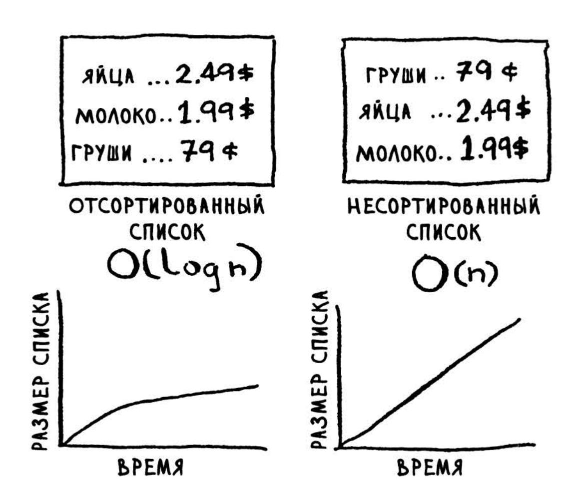


В этой главе

- ✓ Вы узнаете о хеш-таблицах — одной из самых полезных базовых структур данных. Хеш-таблицы находят множество применений; в этой главе рассматриваются распространенные варианты использования.
- ✓ Вы изучите внутреннее устройство хеш-таблиц: реализацию, коллизии и хеш-функции. Это поможет вам понять, как анализируется производительность хеш-таблицы.

Представьте, что вы продавец в маленьком магазинчике. Когда клиент покупает товары, вы проверяете их цену по книге. Если записи в книге не упорядочены по алфавиту, то поиск слова «апельсины» в каждой строке займет слишком много времени. Фактически вам придется проводить простой поиск из главы 1, а для этого нужно проверить каждую запись. Помните, сколько времени это займет? $O(n)$. Если же книга упорядочена по алфавиту, вы сможете воспользоваться бинарным поиском, время которого составляет всего $O(\log n)$.





На всякий случай напомню, что время $O(n)$ и $O(\log n)$ — далеко не одно и то же! Предположим, вы можете просмотреть 10 записей в книге за секунду. В следующей таблице показано, сколько времени займет простой и бинарный поиск.

КОЛИЧЕСТВО ЗАПИСЕЙ В КНИГЕ	O(n)	O(Log n)	
100	10 с	1 с	← НЕОБХОДИМО ПРОВЕРИТЬ $LOG_2 100$ = 7 СТРОК
1000	1.66 МИН	1 с	← НЕОБХОДИМО ПРОВЕРИТЬ $LOG_2 1000$ = 10 СТРОК
10000	16.6 МИН	2 с	← $LOG_2 10000$ = 14 СТРОК =2 С

Вы уже знаете, что бинарный поиск работает очень быстро. Но поиск данных в книге — головная боль для кассира, даже если ее содержимое отсортировано. Пока вы листаете страницы, клиент потихоньку начинает выходить из себя. Гораздо удобнее было бы завести помощницу, которая помнит все названия товаров и цены. Тогда ничего искать вообще не придется: вы спрашиваете помощницу, а она мгновенно отвечает.



Ваша помощница Мэгги может за время $O(1)$ сообщить цену любого товара, независимо от размера книги. Она работает еще быстрее, чем бинарный поиск.

КОЛИЧЕСТВО ЭЛЕМЕНТОВ В КНИГЕ	ПРОСТОЙ ПОИСК $O(n)$	БИНАРНЫЙ ПОИСК $O(\log n)$	МЭГГИ $O(1)$
100	10 с	1 с	МГНОВЕННО
1000	1.6 МИН	1 с	МГНОВЕННО
10000	16.6 МИН	2 с	МГНОВЕННО

Просто чудо, а не девушка! И где взять такую Мэгги?

Обратимся к структурам данных. Пока вам известны две структуры данных: массивы и списки. (О стеках я не говорю, потому что нормальный поиск в стеке невозможен.) Книгу можно реализовать в виде массива.

(ЯЙЦА, 2.49)	(МОЛОКО, 1.49)	(ГРУШИ, 0.79)

---

[1] В русском переводе apple переведено как апельсин, а не как яблоко, чтобы слово начиналось на букву «а». — *Примеч. пер.*

Каждый элемент массива на самом деле состоит из двух элементов: названия товара и его цены. Если отсортировать массив по имени, вы сможете провести по нему бинарный поиск для определения цены товара. Это означает, что поиск будет выполняться за время $O(\log n)$. Но нам нужно, чтобы поиск выполнялся за время $O(1)$ (другими словами, вы хотите создать «Мэгги»). В этом вам помогут хеш-функции.

Хеш-функции

Хеш-функция представляет собой функцию, которая получает строку[1] и возвращает число:



В научной терминологии говорят, что хеш-функция «отображает строки на числа». Можно подумать, что найти закономерности получения чисел для подаваемых на вход строк невозможно. Однако хеш-функция должна соответствовать некоторым требованиям:

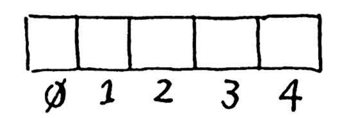
- Она должна быть последовательной. Допустим, вы передали ей строку «апельсины» и получили 4. Это значит, что каждый раз в будущем, передавая ей строку «апельсины», вы будете получать 4. Без этого хеш-таблица бесполезна.
- Разным словам должны соответствовать разные числа. Например, хеш-функция, которая возвращает 1 для каждого полученного слова, никуда

[1] Под «строкой» в данном случае следует понимать любые данные — последовательность байтов.

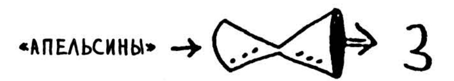
не годится. В идеале каждое входное слово должно отображаться на свое число.

Итак, хеш-функция связывает строки с числами. Зачем это нужно, спросите вы? Так ведь это позволит нам реализовать «Мэгги»!

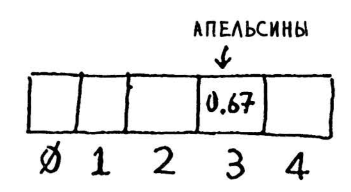
Начнем с пустого массива:



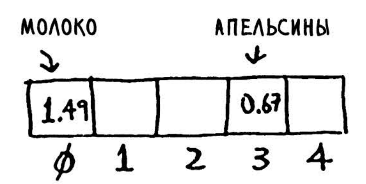
Все цены будут храниться в этом массиве; передадим хеш-функции строку «апельсины».



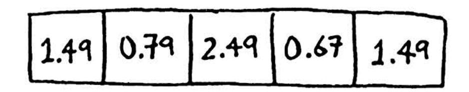
Хеш-функция выдает значение «3». Сохраним цену апельсинов в элементе массива с индексом 3.



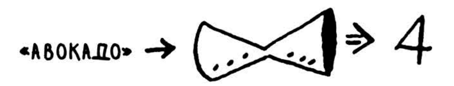
Добавим молоко. Передадим хеш-функции строку «молоко».



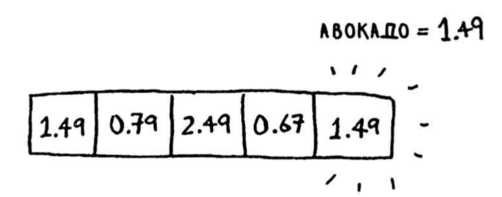
Продолжайте действовать так, и со временем весь массив будет заполнен ценами на товары.



А теперь вы спрашиваете: сколько стоит авокадо? Искать в массиве ничего не нужно, просто передайте строку «авокадо» хеш-функции.



Результат показывает, что значение хранится в элементе с индексом 4. И оно, конечно, там и находится!



Хеш-функция сообщает, где хранится цена, и вам вообще не нужно ничего искать! Такое решение работает, потому что:

- Хеш-функция неизменно связывает название с одним индексом. Каждый раз, когда она вызывается для строки «авокадо», вы получаете обратно одно и то же число. При первом вызове этой функции вы узнаете, где следует сохранить цену авокадо, а при последующих вызовах она сообщает, где взять эту цену.

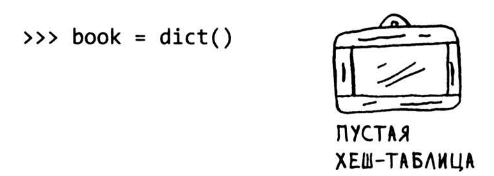
- Хеш-функция связывает разные строки с разными индексами. «Авокадо» связывается с индексом 4, а «молоко» — с индексом 0. Для каждой строки находится отдельная позиция массива, в которой сохраняется цена этого товара.

- Хеш-функция знает размер массива и возвращает только действительные индексы. Таким образом, если длина массива равна 5 элементам, хеш-функция не вернет 100, потому что это значение не является действительным индексом в массиве.

Поздравляю: вы создали «Мэгги»! Свяжите воедино хеш-функцию и массив, и вы получите структуру данных, которая называется *хеш-таблицей*. Хеш-таблица станет первой изученной вами структурой данных, с которой связана дополнительная логика. Массивы и списки напрямую отображаются на адреса памяти, но хеш-таблицы устроены более умно. Они определяют место хранения элементов при помощи хеш-функций.

Вероятно, хеш-таблицы станут самой полезной из сложных структур данных, с которыми вы познакомитесь. Они также известны под другими названиями: «ассоциативные массивы», «словари», «отображения», «хеш-карты» или просто «хеши». Хеш-таблицы исключительно быстро работают! Помните описание массивов и связанных списков из главы 2? Обращение к элементу массива происходит мгновенно. А хеш-таблицы используют массивы для хранения данных, поэтому при обращении к элементам они не уступают массивам.

Скорее всего, вам никогда не придется заниматься реализацией хеш-таблиц самостоятельно. В любом приличном языке существует реализация хеш-таблиц. В Python тоже есть хеш-таблицы; они называются *словарями*. Новая хеш-таблица создается функцией `dict`:



`book` — новая хеш-таблица. Добавим в `book` несколько цен:

```
>>> book["apple"] = 0.67        <-- Апельсины стоят 67 центов
>>> book["milk"] = 1.49         <-- Молоко стоит 1 доллар 49 центов
>>> book["avocado"] = 1.49
>>> print book
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

Пока все просто! А теперь запросим цену авокадо:

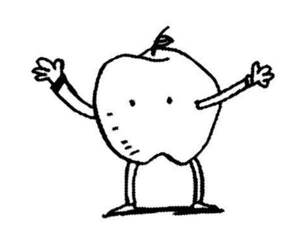
```
>>> print book["avocado"]
1.49      <······················ Цена авокадо
```



Хеш-таблица состоит из ключей и значений. В хеше book имена продуктов являются ключами, а цены — значениями. Хеш-таблица связывает ключи со значениями.

В следующем разделе приведены примеры, в которых хеш-таблицы приносят большую пользу.

Упражнения



Очень важно, чтобы хеш-функции были последовательными, то есть неизменно возвращали один и тот же результат для одинаковых входных данных. Если это условие будет нарушено, вы не сможете найти свой элемент после того, как он будет помещен в хеш-таблицу!

Какие из следующих функций являются последовательными?

5.1 `f(x) = 1` ← Возвращает "1" для любых входных значений

5.2 `f(x) = rand()` ← Возвращает случайное число

5.3 `f(x) = next_empty_slot()` ← Возвращает индекс следующего пустого элемента в хеш-таблице

5.4 `f(x) = len(x)` ← Возвращает длину полученной строки

Примеры использования

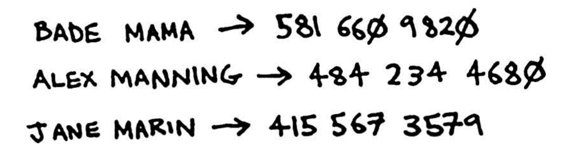
Хеш-таблицы повсеместно применяются на практике. В этом разделе представлены некоторые примеры.

Использование хеш-таблиц для поиска

В вашем телефоне есть удобная встроенная телефонная книга.

С каждым именем связывается номер телефона.





Предположим, вы хотите построить такую телефонную книгу. Имена людей в этой книге связываются с номерами. Телефонная книга должна поддерживать следующие функции:

- добавление имени человека и номера телефона, связанного с этим именем;
- получение номера телефона, связанного с введенным именем.

Такая задача идеально подходит для хеш-таблиц! Хеш-таблицы отлично работают, когда вы хотите:

- создать связь, отображающую один объект на другой;
- найти значение в списке.

Построить телефонную книгу, в общем-то, несложно. Начните с создания новой хеш-таблицы:

```
>>> phone_book = dict()
```

Кстати, в Python предусмотрена сокращенная запись для создания хеш-таблиц: она состоит из двух фигурных скобок:

```
>>> phone_book = {}     ← То же, что phone_book = dict()
```

Добавим в телефонную книгу несколько номеров:

```
>>> phone_book["jenny"] = 8675309
>>> phone_book["emergency"] = 911
```

Вот и все! Теперь предположим, что вы хотите найти номер телефона Дженни (Jenny). Просто передайте ключ хешу:

```
>>> print phone_book["jenny"]
8675309     ← Номер Дженни
```



ХЕШ-ТАБЛИЦА КАК ТЕЛЕФОННАЯ КНИГА

А теперь представьте, что то же самое вам пришлось бы делать с массивом.

Как бы вы это сделали? Хеш-таблицы упрощают моделирование отношений между объектами.

Хеш-таблицы используются для поиска соответствий в гораздо большем масштабе. Например, представьте, что вы хотите перейти на веб-сайт — допустим, *http://adit.io*. Ваш компьютер должен преобразовать символическое имя *adit.io* в IP-адрес.

ADIT.IO → 173.255.248.55

Для любого посещаемого веб-сайта его имя преобразуется в IP-адрес:

GOOGLE.COM → 74.125.239.133
FACEBOOK.COM → 173.252.120.6
SCRIBD.COM → 23.235.47.175

Связать символическое имя с IP-адресом? Идеальная задача для хеш-таблиц! Этот процесс называется *преобразованием DNS*. Хеш-таблицы — всего лишь один из способов реализации этой функциональности.

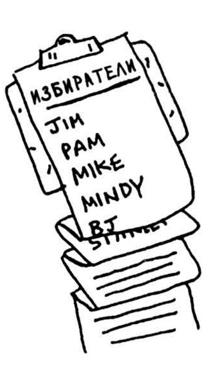
Исключение дубликатов

Предположим, вы руководите избирательным участком. Естественно, каждый избиратель может проголосовать всего один раз. Как проверить, что он не голосовал ранее? Когда человек приходит голосовать, вы узнаете его полное имя, а затем проверяете по списку уже проголосовавших избирателей.





Если имя входит в список, значит, этот человек уже проголосовал — гоните наглеца! В противном случае вы добавляете имя в список и разрешаете ему проголосовать. Теперь предположим, что желающих проголосовать много и список уже проголосовавших достаточно велик.



Каждый раз, когда кто-то приходит голосовать, вы вынуждены просматривать этот гигантский список и проверять, голосовал он или нет. Однако существует более эффективное решение: воспользоваться хешем!

Сначала создадим хеш для хранения информации об уже проголосовавших людях:

```
>>> voted = {}
```

Когда кто-то приходит голосовать, проверьте, присутствует ли его имя в хеше:

```
>>> value = voted.get("tom")
```

Функция `get` возвращает значение, если ключ `"tom"` присутствует в хеш-таблице. В противном случае возвращается `None`. С помощью этой функции можно проверить, голосовал избиратель ранее или нет!



Код выглядит так:

```
voted = {}
def check_voter(name):
  if voted.get(name):
    print "kick them out!"
  else:
    voted[name] = True
    print "let them vote!"
```

Давайте протестируем его на нескольких примерах:

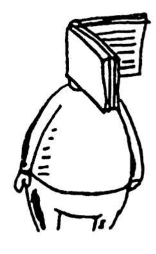
```
>>> check_voter("tom")
let them vote!
>>> check_voter("mike")
let them vote!
>>> check_voter("mike")
kick them out!
```

Когда Том приходит на участок в первый раз, программа разрешает ему проголосовать. Потом приходит Майк, который тоже допускается к голосованию. Но потом Майк делает вторую попытку, и на этот раз у него ничего не получается.

Если бы имена проголосовавших хранились в списке, то выполнение функции со временем замедлилось бы, потому что функции пришлось бы проводить простой поиск по всему списку. Но имена хранятся в хеш-таблице, а хеш-таблица мгновенно сообщает, присутствует имя избирателя в списке или нет. Проверка дубликатов в хеш-таблице выполняется очень быстро.

Использование хеш-таблицы как кэша

Последний пример: кэширование. Если вы работаете над созданием веб-сайтов, вероятно, вы уже слышали о пользе кэширования. Общая идея кэширования такова: допустим, вы заходите на сайт *facebook.com*:

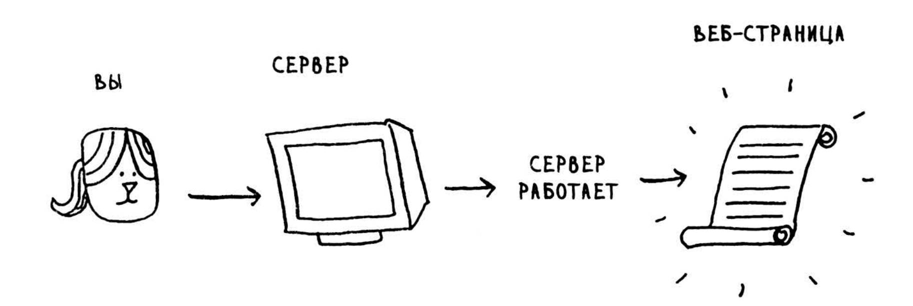




1. Вы обращаетесь с запросом к серверу Facebook.

2. Сервер ненадолго задумывается, генерирует веб-страницу и отправляет ее вам.

3. Вы получаете веб-страницу.

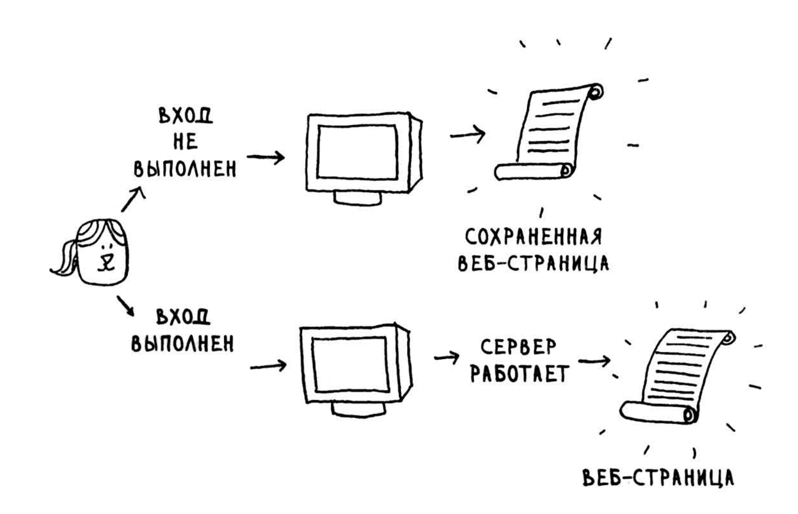


Например, на Facebook сервер может собирать информацию о действиях всех ваших друзей, чтобы представить ее вам. На то, чтобы собрать всю информацию и передать ее вам, требуется пара секунд. С точки зрения пользователя, пара секунд — это очень долго. Он начинает думать: «Почему Facebook работает так медленно?» С другой стороны, серверам Facebook приходится обслуживать миллионы людей, и эти пары секунд для них суммируются. Серверы Facebook трудятся в полную силу, чтобы сгенерировать все эти страницы. Нельзя ли как-то ускорить работу Facebook при том, чтобы серверы выполняли меньше работы?

Представьте, что у вас есть племянница, которая пристает к вам с вопросами о планетах: «Сколько километров от Земли до Марса?», «А сколько километров до Луны?», «А до Юпитера?» Каждый раз вы вводите запрос в Google и сообщаете ей ответ. На это уходит пара минут. А теперь представьте, что она всегда спрашивает: «Сколько километров от Земли до Луны?» Довольно быстро вы запоминаете, что Луна находится на расстоянии 384 400 километров от Земли. Искать информацию в Google не нужно... вы просто запоминаете и выдаете ответ. Вот так работает механизм кэширования: сайт просто запоминает данные, вместо того чтобы пересчитывать их заново.

Если вы вошли на Facebook, то весь контент, который вы видите, адаптирован специально для вас. Каждый раз, когда вы заходите на *facebook.com*, серверам приходится думать, какой контент вас интересует. Если же вы не

ввели учетные данные на Facebook, то вы видите страницу входа. Все пользователи видят одну и ту же страницу входа. Facebook постоянно получает одинаковые запросы: «Я еще не вошел на сайт, выдайте мне домашнюю страницу». Сервер перестает выполнять лишнюю работу и генерировать домашнюю страницу снова и снова. Вместо этого он запоминает, как выглядит домашняя страница, и отправляет ее вам.



Такой механизм хранения называется *кэшированием*. Он обладает двумя преимуществами:

- вы получаете веб-страницу намного быстрее, как и в том случае, когда вы запомнили расстояние от Земли до Луны. Когда племянница в следующий раз задаст вопрос, вам не придется гуглить. Вы можете выдать ответ мгновенно;
- Facebook приходится выполнять меньше работы.

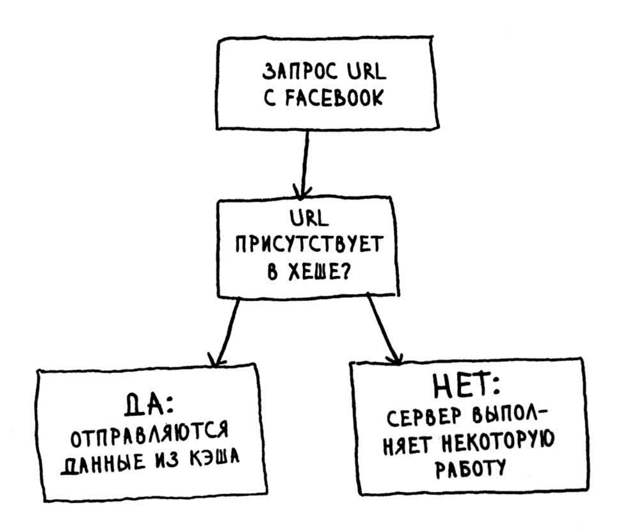
Кэширование — стандартный способ ускорения работы. Все крупные веб-сайты применяют кэширование. А кэшируемые данные хранятся в хеше!

Facebook не просто кэширует домашнюю страницу. Также кэшируются страницы «О нас», «Условия использования» и многие другие. Следовательно, необходимо создать связь URL-адреса страницы и данных страницы.

facebook.com/about → ДАННЫЕ СТРАНИЦЫ С ИНФОРМАЦИЕЙ О FACEBOOK

facebook.com → ДАННЫЕ ДОМАШНЕЙ СТРАНИЦЫ

Когда вы посещаете страницу на сайте Facebook, сайт сначала проверяет, хранится ли страница в хеше.



Вот как это выглядит в коде:

```
cache = {}
def get_page(url):
  if cache.get(url):
    return cache[url]      ◄······ Возвращаются кэшированные данные
  else:
    data = get_data_from_server(url)
    cache[url] = data      ◄······ Данные сначала сохраняются в кэше
    return data
```

Здесь сервер выполняет работу только в том случае, если URL не хранится в кэше. Однако перед тем, как возвращать данные, вы сохраняете их в кэше. Когда пользователь в следующий раз запросит тот же URL-адрес, данные

можно отправить из кэша (вместо того чтобы заставлять сервер выполнять работу).

Шпаргалка

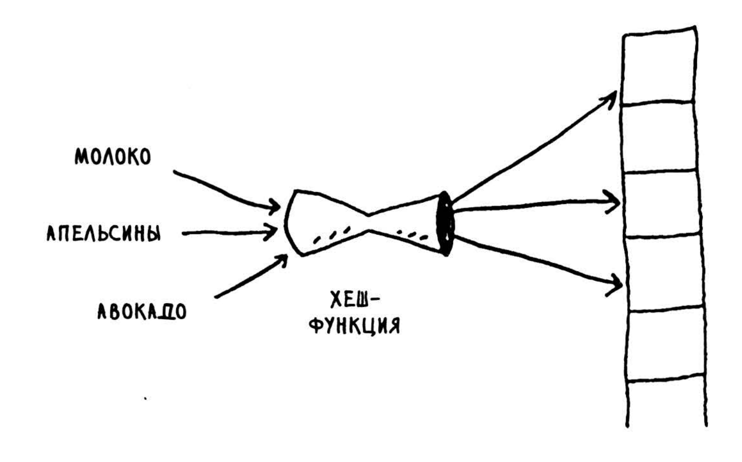
Хеши хорошо подходят для решения следующих задач:

- моделирование отношений между объектами;
- устранение дубликатов;
- кэширование/запоминание данных вместо выполнения работы на сервере.

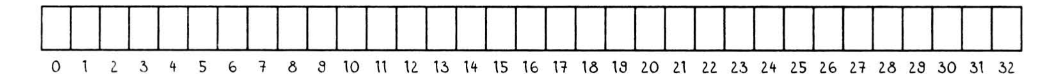
Коллизии

Как я уже сказал, в большинстве языков существуют свои хеш-таблицы. Вам не нужно знать, как написать собственную реализацию, поэтому я не буду надолго останавливаться на внутреннем строении хеш-таблиц. Но быстродействие-то важно всегда! Чтобы понять быстродействие хеш-таблиц, необходимо сначала понять, что такое коллизии. В следующих двух разделах рассматриваются коллизии и быстродействие хеш-таблиц.

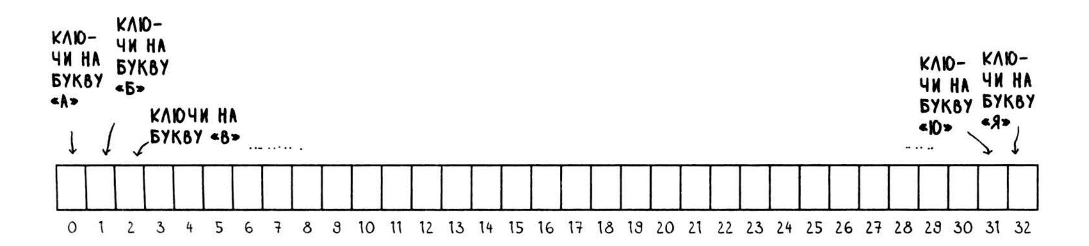
Прежде всего, я немножко приукрасил действительность. Я сказал, что хеш-функция всегда отображает разные ключи на разные позиции в массиве.

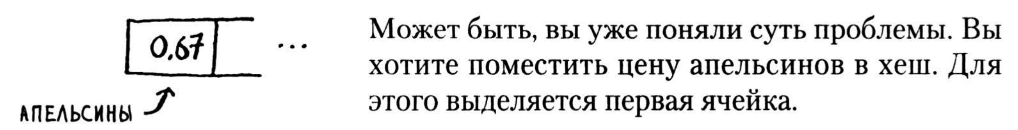


На самом деле написать такую хеш-функцию почти невозможно. Рассмотрим простой пример: допустим, массив состоит всего из 33 ячеек.



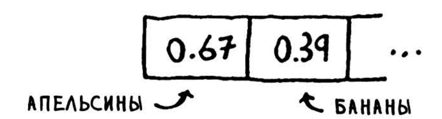
И хеш-функция очень простая: элемент массива просто назначается по алфавитному признаку.



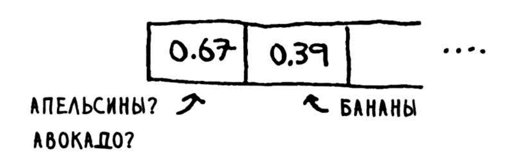


Может быть, вы уже поняли суть проблемы. Вы хотите поместить цену апельсинов в хеш. Для этого выделяется первая ячейка.

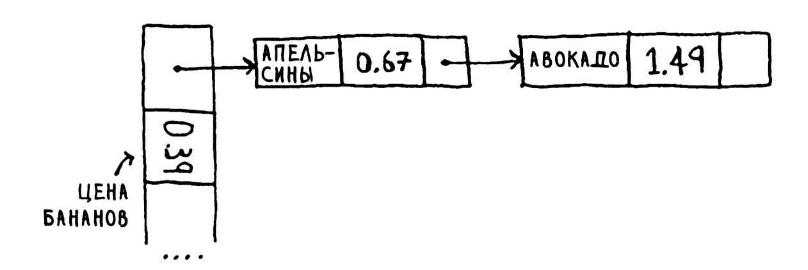
После апельсинов в хеш заносится цена бананов. Для бананов выделяется вторая ячейка.



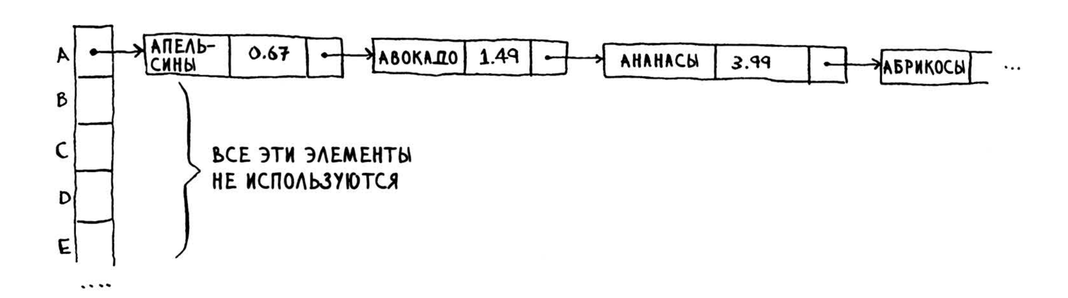
Пока все прекрасно! Но теперь в хеш нужно включить цену авокадо. И для авокадо снова выделяется первая ячейка.



О нет! Элемент уже занят апельсинами! Что же делать? Такая ситуация называется *коллизией*: двум ключам назначается один элемент массива. Возникает проблема: если сохранить в этом элементе цену авокадо, то она запишется на место цены апельсинов. И когда кто-нибудь спросит, сколько стоят апельсины, вы вместо этого сообщите цену авокадо! Коллизии — неприятная штука, и вам придется как-то разбираться с ними. Существует много разных стратегий обработки коллизий. Простейшая из них выглядит так: если несколько ключей отображаются на один элемент, в этом элементе создается связанный список.



В этом примере и «апельсины», и «авокадо» отображаются на один элемент массива, поэтому в элементе создается связанный список. Если вам потребуется узнать цену бананов, эта операция по-прежнему выполнится быстро. Если потребуется узнать цену апельсинов, работа пойдет чуть медленнее. Вам придется провести поиск по связанному списку, чтобы найти в нем «апельсины». Если связанный список мал, это не так страшно — поиск будет ограничен тремя или четырьмя элементами. Но предположим, что вы работаете в специализированной лавке, в которой продаются только продукты на букву «а».



Одну минуту! Вся хеш-таблица полностью пуста, кроме одной ячейки. И эта ячейка содержит огромный связанный список! Каждый элемент этой хеш-таблицы хранится в связанном списке. Ситуация ничуть не лучше той, когда все данные сразу хранятся в связанном списке. Работа с данными замедляется.

Из этого примера следуют два важных урока:

- *выбор хеш-функции действительно важен.* Хеш-функция, отображающая все ключи на один элемент массива, никуда не годится. В идеале хеш-функция должна распределять ключи равномерно по всему хешу;
- если связанные списки становятся слишком длинными, работа с хеш-таблицей сильно замедляется. Но они не станут слишком длинными *при использовании хорошей хеш-функции*!

Хеш-функции играют важную роль. Хорошая хеш-функция создает минимальное число коллизий. Как же выбрать хорошую хеш-функцию? Об этом в следующем разделе!

Быстродействие

Глава началась с примера магазинчика. Вы хотели построить механизм, который мгновенно выдает цены на продукты. Что ж, хеш-таблицы работают очень быстро.

	СРЕДНИЙ СЛУЧАЙ	ХУДШИЙ СЛУЧАЙ
ПОИСК	O(1)	O(n)
ВСТАВКА	O(1)	O(n)
УДАЛЕНИЕ	O(1)	O(n)

БЫСТРОДЕЙСТВИЕ ХЕШ-ТАБЛИЦ

В среднем хеш-таблицы выполняют любые операции за время $O(1)$. Время $O(1)$ называется *постоянным*. Ранее примеры постоянного времени вам еще

не встречались. Оно не означает, что операции выполняются мгновенно; просто время остается постоянным независимо от размера хеш-таблицы. Например, вы знаете, что простой поиск выполняется за линейное время.



Бинарный поиск работает быстрее — за логарифмическое время:



Поиск данных в хеш-таблице выполняется за постоянное время.



Видите горизонтальную линию? Она означает, что при любом размере хеш-таблицы — 1 элемент или 1 миллиард элементов — выборка данных займет одинаковое время. На самом деле вы уже сталкивались с постоянным временем: получение элемента из массива выполняется за постоянное

время. От размера массива оно не зависит. В среднем случае хеш-таблицы работают действительно быстро.

В худшем случае все операции с хеш-таблицей выполняются за время $O(n)$ (линейное время), а это очень медленно. Сравним хеш-таблицы с массивами и списками.

	ХЕШ-ТАБЛИЦЫ (СРЕДНИЙ СЛУЧАЙ)	ХЕШ-ТАБЛИЦЫ (ХУДШИЙ СЛУЧАЙ)	МАССИВЫ	СВЯЗАННЫЕ СПИСКИ
ПОИСК	O(1)	O(n)	O(1)	O(n)
ВСТАВКА	O(1)	O(n)	O(n)	O(1)
УДАЛЕНИЕ	O(1)	O(n)	O(n)	O(1)

Взгляните на средний случай для хеш-таблиц. При поиске хеш-таблицы не уступают в скорости массивам (получение значения по индексу). А при вставке и удалении они так же быстры, как и связанные списки. Получается, что они взяли лучшее от обеих структур! Но в худшем случае хеш-таблицы медленно выполняют все эти операции, поэтому очень важно избегать худшего случая быстродействия при работе с хеш-таблицами. А для этого следует избегать коллизий. Для предотвращения коллизий необходимы:

- низкий коэффициент заполнения;
- хорошая хеш-функция.

ПРИМЕЧАНИЕ

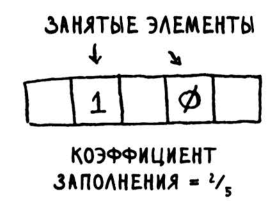
Материал следующего раздела не является обязательным. Речь пойдет о том, как реализовать хеш-таблицу, но вам никогда не придется делать это самостоятельно. Какой бы язык программирования вы ни выбрали, в нем найдется готовая реализация хеш-таблиц. Вы можете воспользоваться встроенной реализацией хеш-таблицы, не сомневаясь в том, что она имеет хорошую эффективность. А в следующем разделе мы заглянем во внутреннее устройство хеш-таблиц.

Коэффициент заполнения

Коэффициент заполнения хеш-таблицы вычисляется по простой формуле.

КОЛИЧЕСТВО ЭЛЕМЕНТОВ В ХЕШ-ТАБЛИЦЕ
———————————
ОБЩЕЕ КОЛИЧЕСТВО ЭЛЕМЕНТОВ

Хеш-таблицы используют массив для хранения данных, поэтому для вычисления коэффициента заполнения можно подсчитать количество заполненных элементов в массиве. Например, в следующей хеш-таблице коэффициент заполнения равен $^2/_5$, или 0,4.

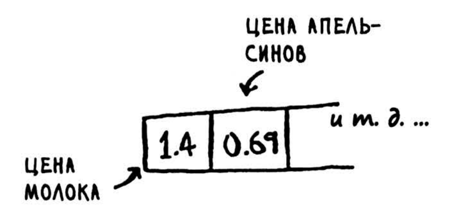


Скажите, каков коэффициент заполнения этой таблицы?



Если вы ответили «$^1/_3$» — все правильно. По коэффициенту заполнения можно оценить количество пустых ячеек в хеш-таблице.

Предположим, в хеш-таблице нужно сохранить цены 100 товаров и хеш-таблица состоит из 100 элементов. В лучшем случае каждому товару будет выделен отдельный элемент.



Коэффициент заполнения этой хеш-таблицы равен 1. А если хеш-таблица состоит всего из 50 элементов? Тогда ее коэффициент заполнения будет равен 2. Выделить под каждый товар отдельный элемент ни при каких условиях не удастся, потому что элементов попросту не хватит! Коэффициент заполнения больше 1 означает, что количество товаров превышает количество элементов в массиве.

С ростом коэффициента заполнения в хеш-таблицу приходится добавлять новые элементы, то есть изменять ее размер. Представим, что эта хеш-таблица приближается к заполнению.



Хеш-таблицу необходимо расширить. Расширение начинается с создания нового массива большего размера. Обычно в таком случае создается массив вдвое большего размера.



Теперь все эти элементы необходимо заново вставить в новую хеш-таблицу функцией hash:

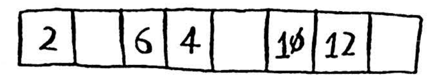


Новая таблица имеет коэффициент заполнения $^3/_8$. Гораздо лучше! С меньшим коэффициентом загрузки число коллизий уменьшается, и ваша таблица начинает работать более эффективно. Хорошее приближенное правило:

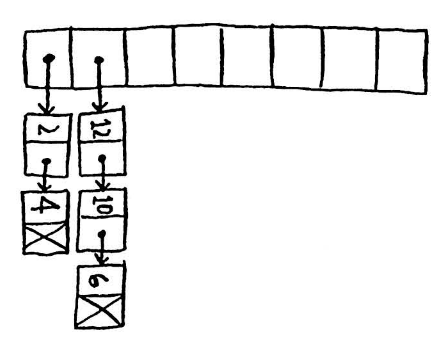
изменяйте размер хеш-таблицы, когда коэффициент заполнения превышает 0,7. Но ведь на изменение размеров уходит много времени, скажете вы, и будете абсолютно правы! Да, изменение размеров требует значительных затрат ресурсов, поэтому оно не должно происходить слишком часто. В среднем хеш-таблицы работают за время $O(1)$ даже с изменением размеров.

Хорошая хеш-функция

Хорошая хеш-функция должна обеспечивать равномерное распределение значений в массиве.



Плохая хеш-функция создает скопления и порождает множество коллизий.



Какую хеш-функцию считать хорошей? К счастью, вам об этом никогда не придется беспокоиться — пусть об этом беспокоятся пожилые бородатые умники, сидящие в полутемных комнатах. Если вам интересна эта тема, поищите информацию об алгоритме SHA (короткое описание приведено в последней главе). Вы можете использовать этот алгоритм в своей хеш-функции.

Упражнения

Очень важно, чтобы хеш-функции обеспечивали хорошее распределение. Они должны распределять значения как можно шире. Худший случай — хеш-функция, которая отображает все значения на одну позицию в хеш-таблице.

Предположим, имеются четыре хеш-функции, которые получают строки:

1. Первая функция возвращает «1» для любого входного значения.
2. Вторая функция возвращает длину строки в качестве индекса.
3. Третья функция возвращает первый символ строки в качестве индекса. Таким образом, все строки, начинающиеся с «a», хешируются в одну позицию, все строки, начинающиеся с «b» — в другую и т. д.
4. Четвертая функция ставит в соответствие каждой букве простое число: a = 2, b = 3, c = 5, d = 7, e = 11 и т. д. Для строки хеш-функцией становится остаток от деления суммы всех значений на размер хеша. Например, если размер хеша равен 10, то для строки «bag» будет вычислен индекс 3+2+17%10 = 22%10 = 2.

В каком из этих примеров хеш-функции будут обеспечивать хорошее распределение? Считайте, что хеш-таблица содержит 10 элементов.

5.5 Телефонная книга, в которой ключами являются имена, а значениями – номера телефонов. Задан следующий список имен: Esther, Ben, Bob, Dan.

5.6 Связь размера батарейки с напряжением. Размеры батареек: A, AA, AAA, AAAA.

5.7 Связь названий книг с именами авторов. Названия книг: «Maus», «Fun Home», «Watchmen».

Шпаргалка

Вам почти никогда не придется реализовать хеш-таблицу самостоятельно. Язык программирования, который вы используете, должен предоставить необходимую реализацию. Вы можете пользоваться хеш-таблицами Python, и при этом вам будет обеспечена производительность среднего случая: постоянное время.

Хеш-таблицы чрезвычайно полезны, потому что они обеспечивают высокую скорость операций и позволяют по-разному моделировать данные. Возможно, вскоре выяснится, что вы постоянно используете их в своей работе.

- Хеш-таблица создается объединением хеш-функции с массивом.
- Коллизии нежелательны. Хеш-функция должна свести количество коллизий к минимуму.
- Хеш-таблицы обеспечивают очень быстрое выполнение поиска, вставки и удаления.
- Хеш-таблицы хорошо подходят для моделирования отношений между объектами.
- Как только коэффициент заполнения превышает 0,7, пора изменять размер хеш-таблицы.
- Хеш-таблицы используются для кэширования данных (например, на веб-серверах).
- Хеш-таблицы хорошо подходят для обнаружения дубликатов.



6 Поиск в ширину



В этой главе

- ✓ Вы научитесь моделировать сети при помощи новой абстрактной структуры данных — графов.
- ✓ Вы освоите поиск в ширину — алгоритм, который применяется к графам для получения ответов на вопросы вида «Какой кратчайший путь ведет к X?»
- ✓ Вы узнаете, чем направленные графы отличаются от ненаправленных.
- ✓ Вы освоите топологическую сортировку — другой алгоритм сортировки, раскрывающий связи между узлами.

Эта глава посвящена графам. Сначала вы узнаете, что такое граф. Затем я покажу первый алгоритм, работающий с графами. Он называется *поиском в ширину* (BFS, Breadth-First Search).

Поиск в ширину позволяет найти кратчайшее расстояние между двумя объектами. Однако сам термин «кратчайшее расстояние» может иметь много разных значений! Например, с помощью поиска в ширину можно:

- ❑ написать программу для игры в шашки, которая вычисляет кратчайший путь к победе;

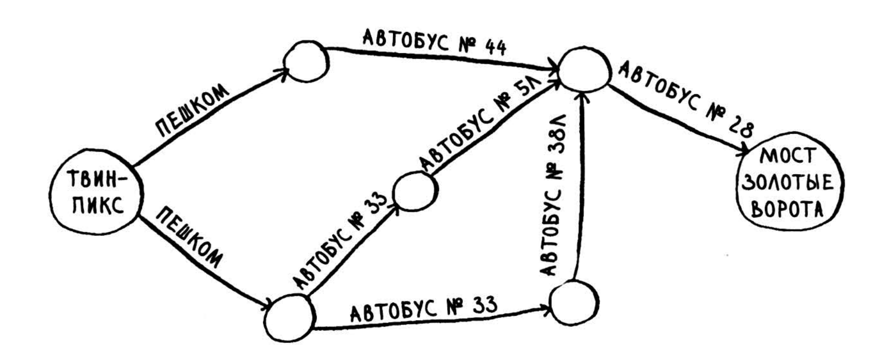
- реализовать проверку правописания (минимальное количество изменений, преобразующих ошибочно написанное слово в правильное, например АЛГОРИФМ -> АЛГОРИТМ — одно изменение);
- найти ближайшего к вам врача.

Одни из самых полезных алгоритмов, известных мне, работают с графами. Внимательно прочитайте несколько следующих глав — этот материал неоднократно пригодится вам в работе.

Знакомство с графами

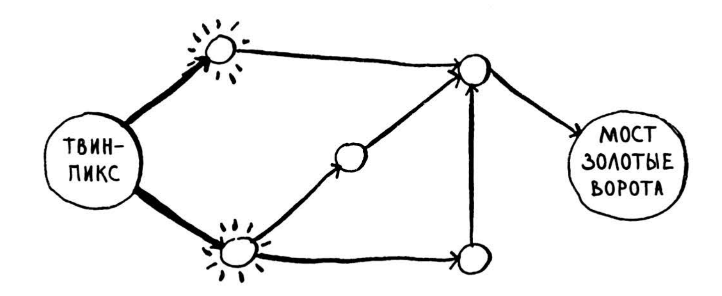


Предположим, вы находитесь в Сан-Франциско и хотите добраться из Твин-Пикс к мосту Золотые Ворота. Вы намереваетесь доехать на автобусе с минимальным количеством пересадок. Возможные варианты:

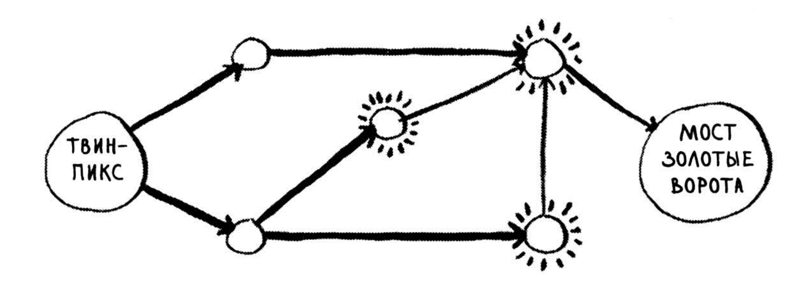


Какой алгоритм вы бы использовали для поиска пути с наименьшим количеством шагов?

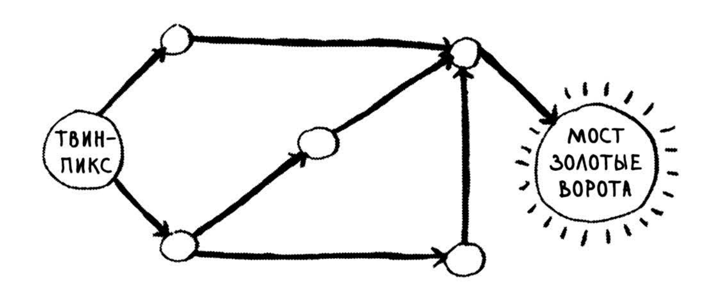
Можно ли сделать это за один шаг? На следующем рисунке выделены все места, в которые можно добраться за один шаг.



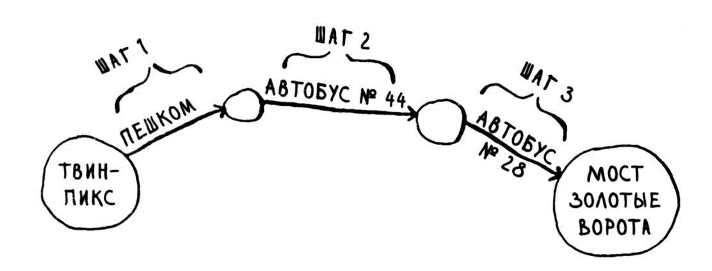
Мост на этой схеме не выделен; до него невозможно добраться за один шаг. А можно ли добраться до него за два шага?



И снова мост не выделен, а значит, до него невозможно добраться за два шага. Как насчет трех шагов?



Ага! На этот раз мост Золотые Ворота выделен. Следовательно, чтобы добраться из Твин-Пикс к мосту по этому маршруту, необходимо сделать три шага.



Есть и другие маршруты, которые приведут вас к мосту, но они длиннее (четыре шага). Алгоритм обнаружил, что кратчайший путь к мосту состоит из трех шагов. Задача такого типа называется *задачей поиска кратчайшего пути*. Часто требуется найти некий кратчайший путь: путь к дому вашего друга, путь к победе в шахматной партии (за наименьшее количество ходов) и т. д. Алгоритм для решения задачи поиска кратчайшего пути называется *поиском в ширину*.

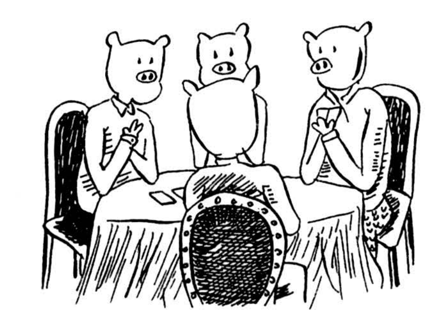
Чтобы найти кратчайший путь из Твин-Пикс к мосту Золотые Ворота, нам пришлось выполнить два шага:

1. Смоделировать задачу в виде графа.

2. Решить задачу методом поиска в ширину.

В следующем разделе я расскажу, что такое графы. Затем будет рассмотрен более подробно поиск в ширину.

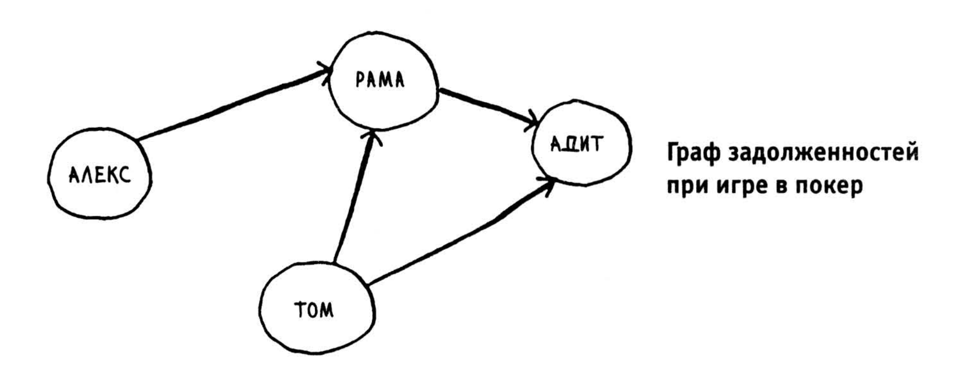
Что такое граф?



Граф моделирует набор связей. Представьте, что вы с друзьями играете в покер и хотите смоделировать, кто кому сейчас должен. Например, условие «Алекс должен Раме» можно смоделировать так:

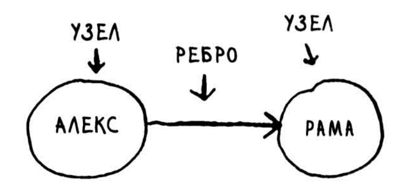


А полный граф может выглядеть так:



Граф задолженностей при игре в покер

Алекс должен Раме, Том должен Адиту и т. д. Каждый граф состоит из *узлов* и *ребер*.



Вот и все! Графы состоят из узлов и ребер. Узел может быть напрямую соединен с несколькими другими узлами. Эти узлы называются *соседями*. На этом графе Рама является соседом Алекса. С другой стороны, Адит соседом Алекса не является, потому что они не соединены напрямую. При этом Адит является соседом Рамы и Тома.

Графы используются для моделирования связей между разными объектами. А теперь посмотрим, как работает поиск в ширину.

Поиск в ширину

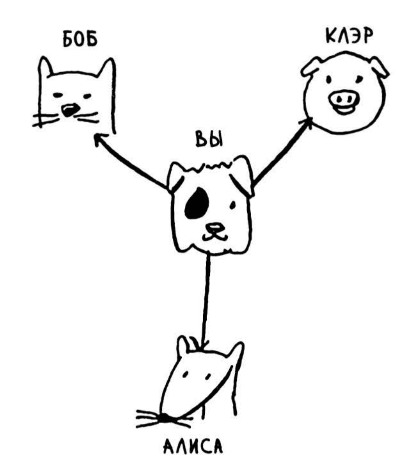
В главе 1 уже рассматривался пример алгоритма поиска: бинарный поиск. Поиск в ширину также относится к категории алгоритмов поиска, но этот алгоритм работает с графами. Он помогает ответить на вопросы двух типов:

- тип 1: существует ли путь от узла A к узлу B?
- тип 2: как выглядит кратчайший путь от узла A к узлу B?

Вы уже видели пример поиска в ширину, когда мы просчитывали кратчайший путь из Твин-Пикс к мосту Золотые Ворота. Это был вопрос типа 2: как выглядит кратчайший путь? Теперь разберем работу алгоритма более подробно с вопросом типа 1: существует ли путь?



Представьте, что вы выращиваете манго. Вы ищете продавца, который будет продавать ваши замечательные манго. А может, продавец найдется среди ваших контактов на Facebook? Для начала стоит поискать среди друзей.

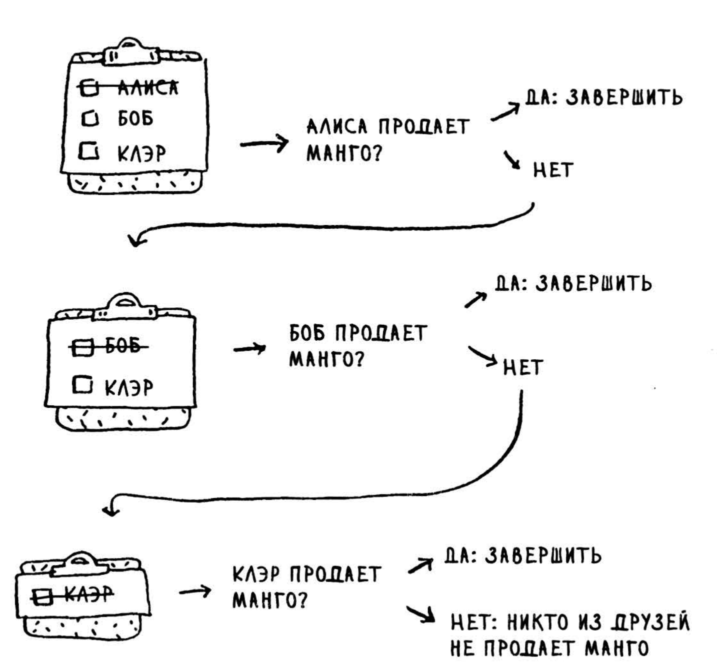


Поиск происходит вполне тривиально.

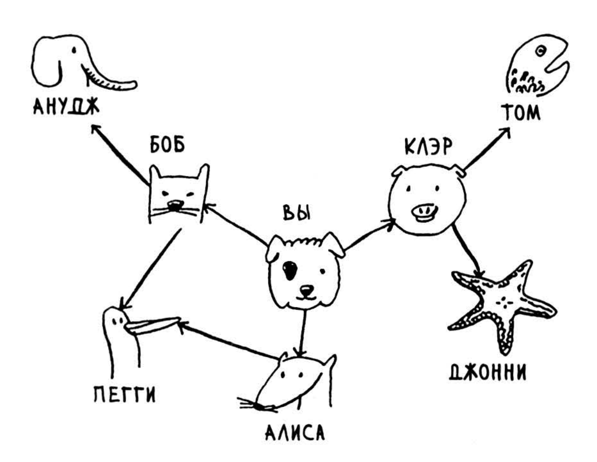
Сначала нужно построить список друзей для поиска.

Теперь нужно обратиться к каждому человеку в списке и проверить, продает ли этот человек манго.

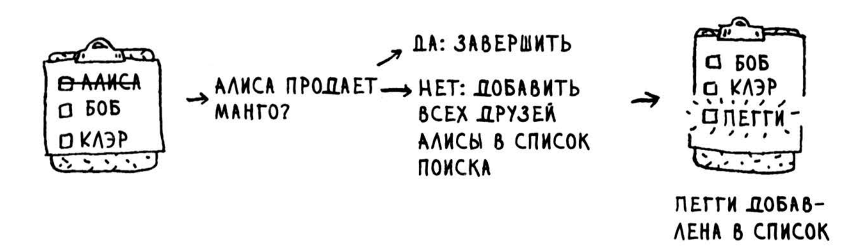




Предположим, ни один из ваших друзей не продает манго. Теперь поиск продолжается среди друзей ваших друзей.



Каждый раз, когда вы проверяете кого-то из списка, вы добавляете в список всех его друзей.



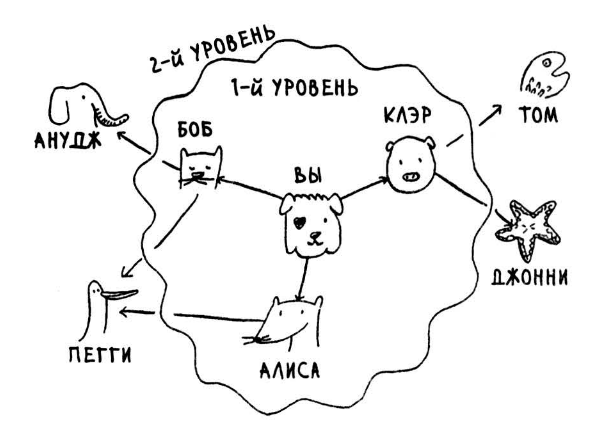
В таком случае поиск ведется не только среди друзей, но и среди друзей друзей тоже. Напомним: нужно найти в сети хотя бы одного продавца манго. Если Алиса не продает манго, то в список добавляются ее друзья. Это означает, что со временем вы проверите всех ее друзей, а потом их друзей и т. д. С этим алгоритмом поиск рано или поздно пройдет по всей сети, пока вы все-таки не наткнетесь на продавца манго. Такой алгоритм и называется поиском в ширину.

Поиск кратчайшего пути

На всякий случай напомню два вопроса, на которые может ответить алгоритм поиска в ширину:

- тип 1: существует ли путь от узла A к узлу B? (Есть ли продавец манго в вашей сети?)
- тип 2: как выглядит кратчайший путь от узла A к узлу B? (Кто из продавцов манго находится ближе всего к вам?)

Вы уже знаете, как ответить на вопрос 1; теперь попробуем ответить на вопрос 2. Удастся ли вам найти ближайшего продавца манго? Будем считать, что ваши друзья — это связи первого уровня, а друзья друзей — связи второго уровня.



Связи первого уровня предпочтительнее связей второго уровня, связи второго уровня предпочтительнее связей третьего уровня и т. д. Отсюда следует, что поиск по контактам второго уровня не должен производиться, пока вы не будете полностью уверены в том, что среди связей первого уровня нет ни одного продавца манго. Но ведь поиск в ширину именно это и делает! Поиск в ширину распространяется от начальной точки. А это означает, что связи первого уровня будут проверены до связей второго уровня. Контрольный вопрос: кто будет проверен первым, Клэр или Ануддж? Ответ:

Клэр является связью первого уровня, а Ануджа — связью второго уровня. Следовательно, Клэр будет проверена первой.

Также можно объяснить это иначе: связи первого уровня добавляются в список поиска раньше связей второго уровня.

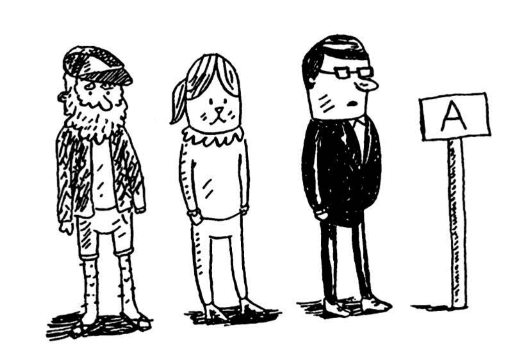


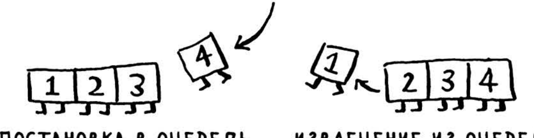
Вы двигаетесь вниз по списку и проверяете каждого человека (является ли он продавцом манго). Связи первого уровня будут проверены до связей второго уровня, так что вы найдете продавца манго, ближайшего к вам. Поиск в ширину находит не только путь из A в B, но и кратчайший путь.

Обратите внимание: это условие выполняется только в том случае, если поиск осуществляется в порядке добавления людей. Другими словами, если Клэр была добавлена в список до Ануджа, то проверка Клэр должна быть выполнена до проверки Ануджа. А что произойдет, если вы проверите Ануджа раньше, чем Клэр, и оба они окажутся продавцами манго? Ануджа является связью второго уровня, а Клэр — связью первого уровня. В результате будет найден продавец манго, не ближайший к вам в сети. Следовательно, проверять связи нужно в порядке их добавления. Для операций такого рода существует специальная структура данных, которая называется *очередью*.

Очереди

Очередь работает точно так же, как и в реальной жизни. Предположим, вы с другом стоите в очереди на автобусной остановке. Если вы стоите ближе к началу очереди, то вы первым сядете в автобус. Структура данных очереди работает аналогично. Очереди чем-то похожи на стеки: вы не можете обращаться к произвольным элементам очереди. Вместо этого поддерживаются всего две операции: *постановка в очередь* и *извлечение из очереди*.





Если вы поставите в очередь два элемента, то элемент, добавленный первым, будет извлечен из очереди раньше второго. А ведь это свойство можно использовать для реализации списка поиска! Люди, добавленные в список первыми, будут извлечены из очереди и проверены первыми.

Очередь относится к категории структур данных FIFO: First In, First Out («первым вошел, первым вышел»). А стек принадлежит к числу структур данных LIFO: Last In, First Out («последним пришел, первым вышел»).

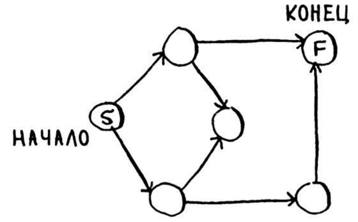


Теперь, когда вы знаете, как работает очередь, можно переходить к реализации поиска в ширину!

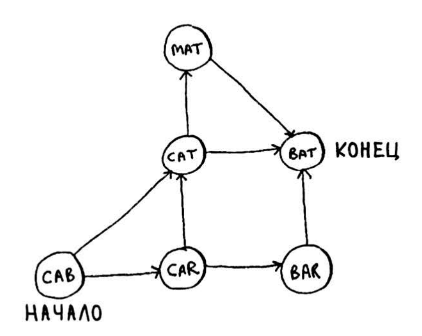
Упражнения

Примените алгоритм поиска в ширину к каждому из этих графов, чтобы найти решение.

6.1 Найдите длину кратчайшего пути от начального до конечного узла.

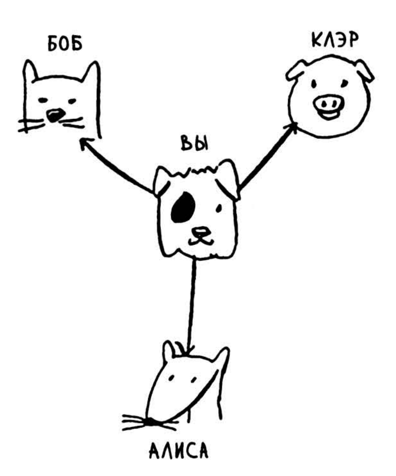


6.2 Найдите длину кратчайшего пути от «cab» к «bat».



Реализация графа

Для начала необходимо реализовать граф на программном уровне. Граф состоит из нескольких узлов. И каждый узел соединяется с соседними узлами. Как выразить отношение типа «вы –> боб»? К счастью, вам уже известна структура данных, способная выражать отношения: *хеш-таблица*!



Вспомните: хеш-таблица связывает ключ со значением. В данном случае узел должен быть связан со всеми его соседями.

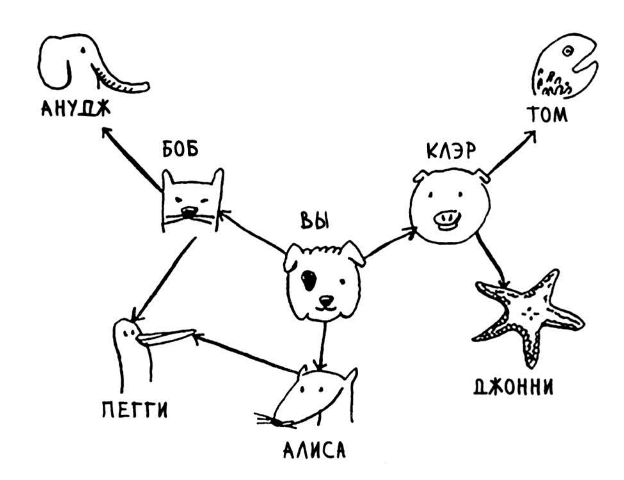


А вот как это записывается на Python:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

Обратите внимание: элемент «вы» (you) отображается на массив. Следовательно, результатом выражения `graph["you"]` является массив всех ваших соседей.

Граф — всего лишь набор узлов и ребер, поэтому для представления графа на Python ничего больше не потребуется. А как насчет большего графа, например такого?



Код на языке Python выглядит так:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
graph["bob"] = ["anuj", "peggy"]
graph["alice"] = ["peggy"]
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
graph["peggy"] = []
graph["thom"] = []
graph["jonny"] = []
```

Контрольный вопрос: важен ли порядок добавления пар «ключ—значение»?

Важно ли, какую запись вы будете использовать, — такую:

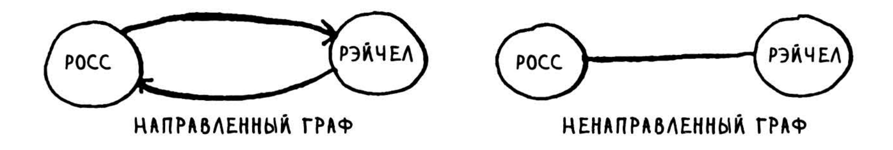
```
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
```

или такую:

```
graph["anuj"] = []
graph["claire"] = ["thom", "jonny"]
```

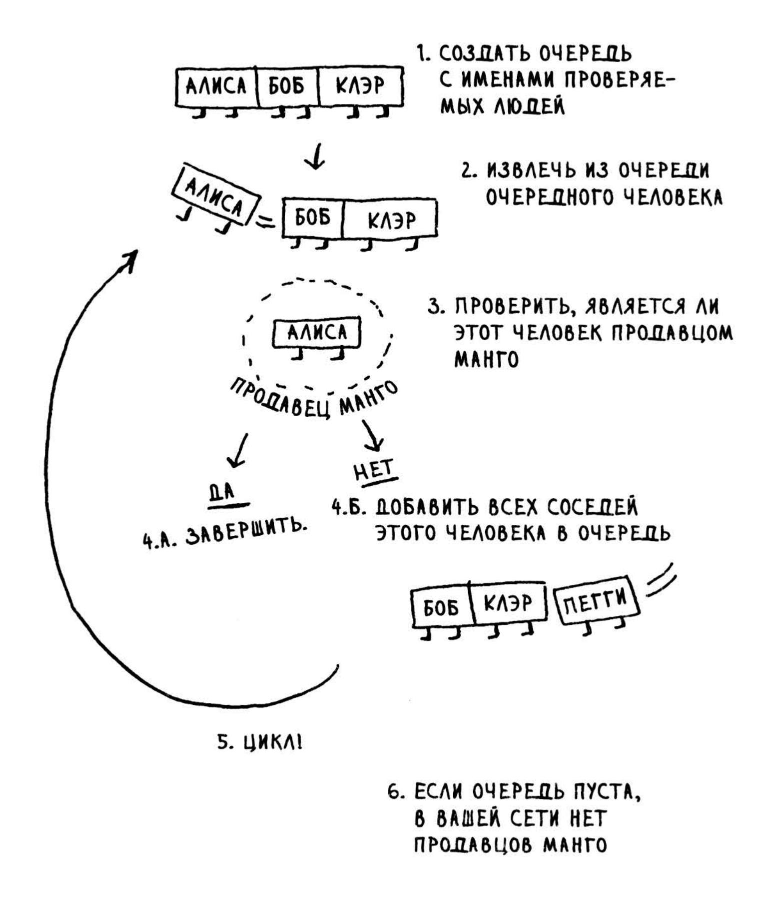
Вспомните предыдущую главу. Ответ: нет, не важно. В хеш-таблицах элементы не упорядочены, поэтому добавлять пары «ключ—значение» можно в любом порядке.

У Ануджа, Пегги, Тома и Джонни соседей нет. Линии со стрелками указывают на них, но не существует стрелок от них к другим узлам. Такой граф называется *направленным* — отношения действуют только в одну сторону. Итак, Анудж является соседом Боба, но Боб не является соседом Ануджа. В ненаправленном графе стрелок нет, и каждый из узлов является соседом по отношению друг к другу. Например, оба следующих графа эквивалентны.



Реализация алгоритма

Напомню, как работает реализация.



Все начинается с создания очереди. В Python для создания *двусторонней очереди* (дека) используется функция deque:

```
from collections import deque
search_queue = deque()          ◄············ Создание новой очереди
search_queue += graph["you"]    ◄············ Все соседи добавляются в очередь поиска
```

Напомню, что выражение graph["you"] вернет список всех ваших соседей, например ["alice", "bob", "claire"]. Все они добавляются в очередь поиска.
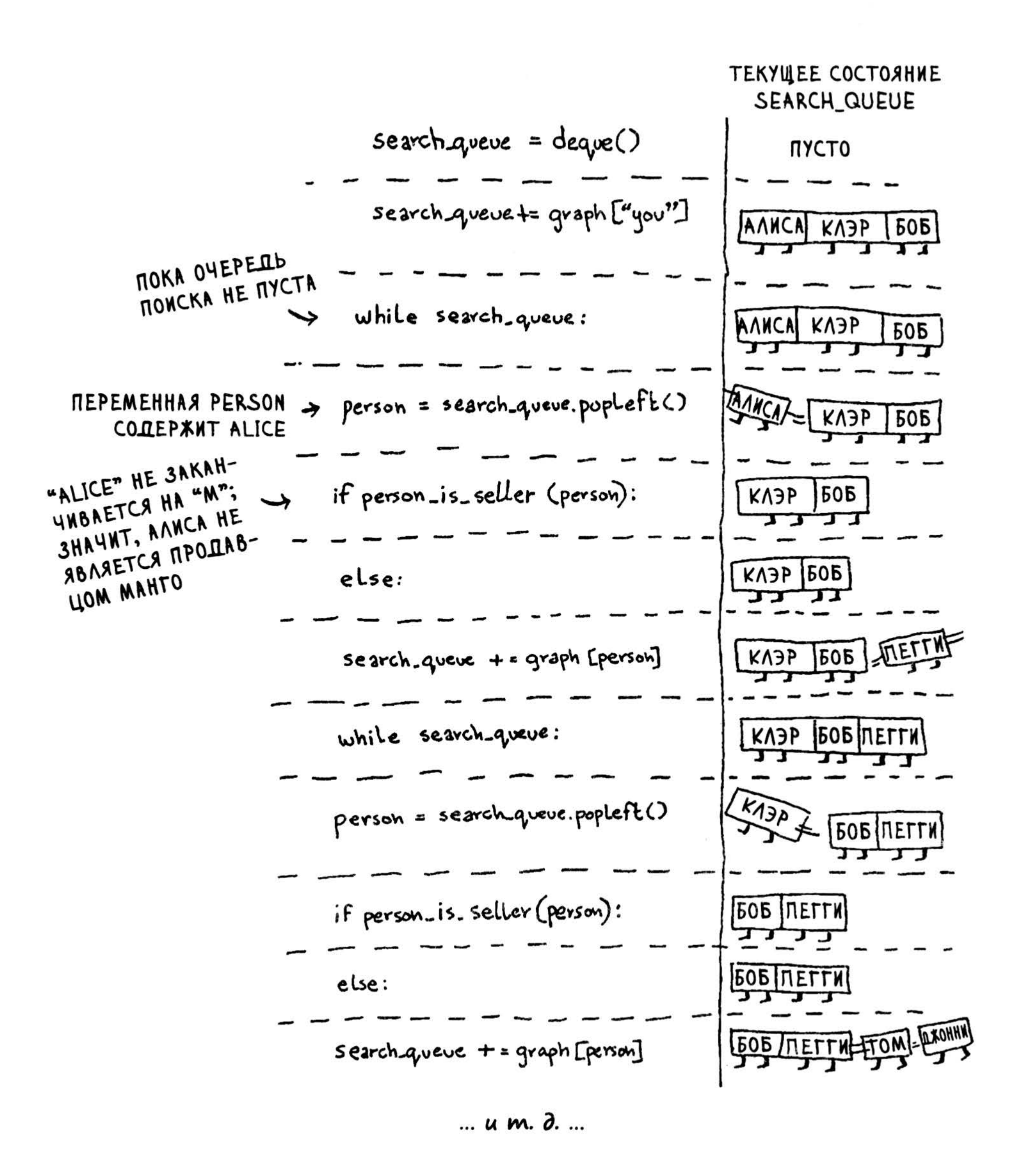


А теперь рассмотрим остальное:

```
while search_queue:              ← Пока очередь не пуста...
    person = search_queue.popleft()   ← из очереди извлекается первый человек
    if person_is_seller(person):  ← Проверяем, является ли этот человек
                                    продавцом манго
        print person + " is a mango seller!"   ← Да, это продавец манго
        return True
    else:
        search_queue += graph[person]   ← Нет, не является. Все друзья этого че-
                                          ловека добавляются в очередь поиска
return False    ← Если выполнение дошло
                  до этой строки, значит,
                  в очереди нет продавца манго
```

И последнее: нужно определить функцию person_is_seller, которая сообщает, является ли человек продавцом манго. Например, функция может выглядеть так:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

Эта функция проверяет, заканчивается ли имя на букву «m», и если заканчивается, этот человек считается продавцом манго. Проверка довольно глупая, но для нашего примера сойдет. А теперь посмотрим, как работает поиск в ширину.



И так далее. Алгоритм продолжает работать до тех пор, пока:

- не будет найден продавец манго,

или

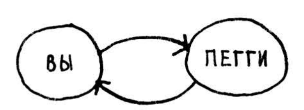
- очередь не опустеет (в этом случае продавца манго нет).

У Алисы и Боба есть один общий друг: Пегги. Следовательно, Пегги будет добавлена в очередь дважды: при добавлении друзей Алисы и при добавлении друзей Боба. В результате Пегги появится в очереди поиска в двух экземплярах.



Но проверить, является ли Пегги продавцом манго, достаточно всего один раз. Проверяя ее дважды, вы выполняете лишнюю, ненужную работу. Следовательно, после проверки человека нужно пометить как проверенного, чтобы не проверять его снова.

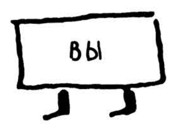
Если этого не сделать, может возникнуть бесконечный цикл. Предположим, граф выглядит так:



В начале очередь поиска содержит всех ваших соседей.



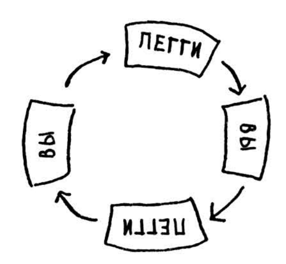
Теперь вы проверяете Пегги. Она не является продавцом манго, поэтому все ее соседи добавляются в очередь поиска.



Вы проверяете себя. Вы не являетесь продавцом манго, поэтому все ваши соседи добавляются в очередь поиска.



И так далее. Возникает бесконечный цикл, потому что очередь поиска будет поочередно переходить от вас к Пегги.



Прежде чем проверять человека, следует убедиться в том, что он не был проверен ранее. Для этого мы будем вести список уже проверенных людей.



А вот окончательная версия кода поиска в ширину, в которой учтено это обстоятельство:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []                    ← Этот массив используется для отслеживания уже проверенных людей
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:   ← Человек проверяется только в том случае, если он не проверялся ранее
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)    ← Человек помечается как уже проверенный
    return False

search("you")
```

Попробуйте выполнить этот код самостоятельно. Замените функцию `person_is_seller` чем-то более содержательным и посмотрите, выведет ли она то, что вы ожидали.

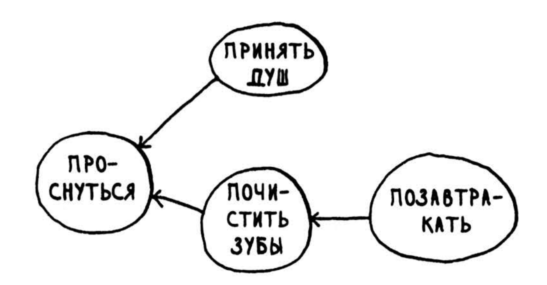
Время выполнения

Если поиск продавца манго был выполнен по всей сети, значит, вы прошли по каждому ребру (напомню: ребром называется соединительная линия или линия со стрелкой, ведущая от одного человека к другому). Таким образом, время выполнения составляет как минимум *O*(*количество ребер*).

Также в программе должна храниться очередь поиска. Добавление одного человека в очередь выполняется за постоянное время: $O(1)$. Выполнение операции для каждого человека потребует суммарного времени *O*(*количество людей*). Поиск в ширину выполняется за время *O*(*количество людей + количество ребер*), что обычно записывается в форме $O(V+E)$ (V — количество вершин, E — количество ребер).

Упражнения

Перед вами небольшой граф моего утреннего распорядка.



Из графа видно, что я завтракаю только после того, как почищу зубы. Таким образом, узел «Позавтракать» зависит от узла «Почистить зубы».

С другой стороны, душ не зависит от чистки зубов, потому что я могу сначала принять душ, а потом почистить зубы. На основании графа можно сформулировать порядок, в котором я действую утром:

1. Проснуться.
2. Принять душ.
3. Почистить зубы.
4. Позавтракать.

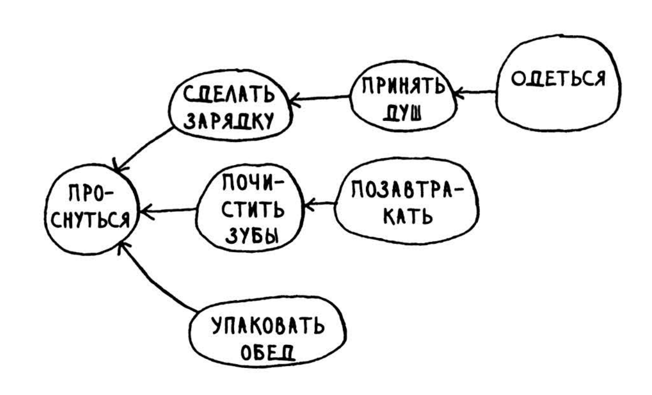
Следует заметить, что действие «Принять душ» может перемещаться в списке, поэтому следующий список тоже действителен:

1. Проснуться.
2. Почистить зубы.
3. Принять душ.
4. Позавтракать.

6.3 Для каждого из следующих трех списков укажите, действителен он или недействителен.

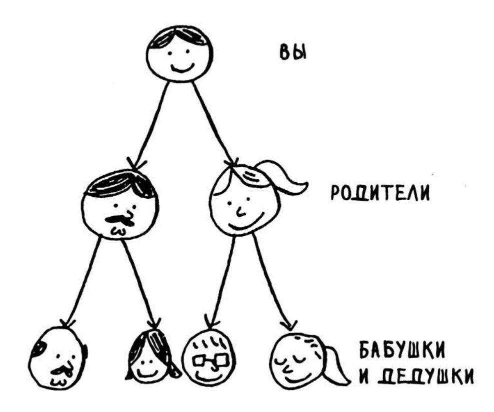
А	Б	В
1. ПРОСНУТЬСЯ	1. ПРОСНУТЬСЯ	1. ПРИНЯТЬ ДУШ
2. ПРИНЯТЬ ДУШ	2. ПОЧИСТИТЬ ЗУБЫ	2. ПРОСНУТЬСЯ
3. ПОЗАВТРАКАТЬ	3. ПОЗАВТРАКАТЬ	3. ПОЧИСТИТЬ ЗУБЫ
4. ПОЧИСТИТЬ ЗУБЫ	4. ПРИНЯТЬ ДУШ	4. ПОЗАВТРАКАТЬ

6.4 Немного увеличим исходный граф. Постройте действительный список для этого графа.

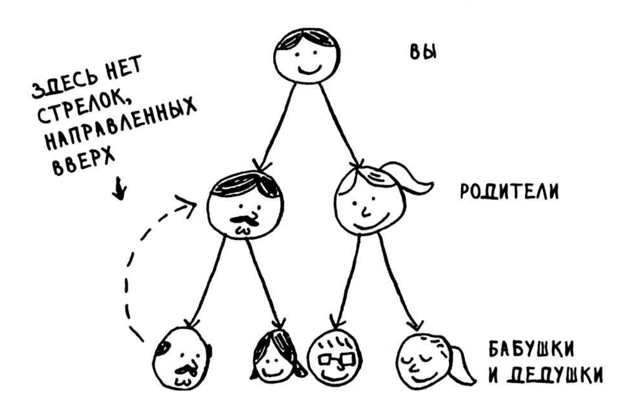


Можно сказать, что этот список в некотором смысле отсортирован. Если задача А зависит от задачи В, то задача А находится в более поздней позиции списка. Такая сортировка называется *топологической*; фактически она предоставляет способ построения упорядоченного списка на основе графа. Предположим, вы планируете свадьбу и у вас составлен большой граф с множеством задач, но вы не знаете, с чего начать. Проведите *топологическую сортировку* графа — и полу́чите список задач, которые можно выполнять одну за другой.

Допустим, имеется генеалогическое древо.

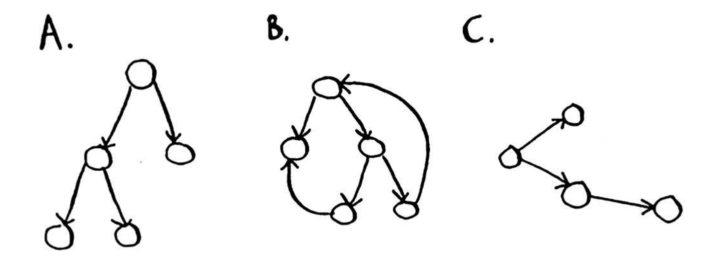


Генеалогическое древо — тоже граф, потому что в нем есть узлы (люди) и ребра. Ребра указывают на родителей человека. Естественно, все ребра направлены вниз — в генеалогическом дереве ребро, указывающее вверх, не имеет смысла. Ведь ваш отец никак не может быть дедушкой вашего дедушки!



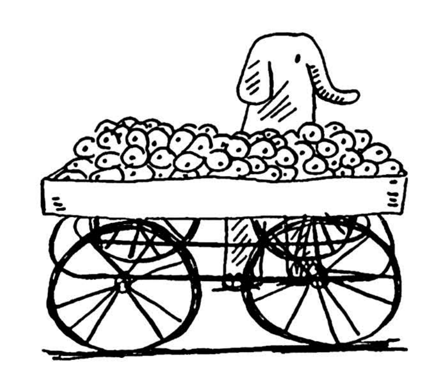
Такая особая разновидность графа, в которой нет ребер, указывающих в обратном направлении, называется *деревом*.

6.5 Какие из следующих графов также являются деревьями?



Шпаргалка

- Поиск в ширину позволяет определить, существует ли путь из A в B.
- Если путь существует, то поиск в ширину находит кратчайший путь.
- Если в вашей задаче требуется найти «кратчайшее X», попробуйте смоделировать свою задачу графом и воспользуйтесь поиском в ширину для ее решения.
- В направленном графе есть стрелки, а отношения действуют в направлении стрелки (Рама → Адит означает «Рама должен Адиту»).
- В ненаправленных графах стрелок нет, а отношение идет в обе стороны (Росс – Рэйчел означает «Росс встречается с Рэйчел, а Рэйчел встречается с Россом».)
- Очереди относятся к категории FIFO («первым вошел, первым вышел»).
- Стек относится к категории LIFO («последним пришел, первым вышел»).
- Людей следует проверять в порядке их добавления в список поиска, поэтому список поиска должен быть оформлен в виде очереди, иначе найденный путь не будет кратчайшим.
- Позаботьтесь о том, чтобы уже проверенный человек не проверялся заново, иначе может возникнуть бесконечный цикл.



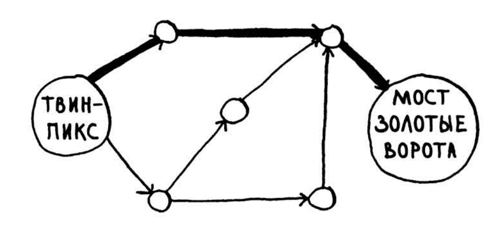
7 Алгоритм Дейкстры



В этой главе

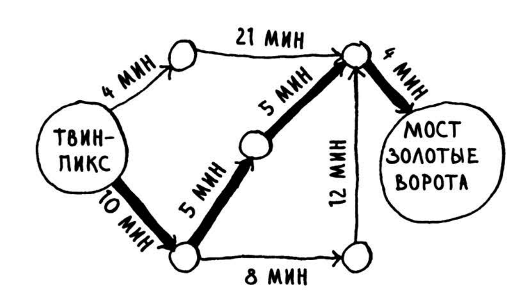
- ✓ Мы продолжим изучение графов и познакомимся со взвешенными графами, в которых некоторым ребрам назначаются большие или меньшие веса.
- ✓ Вы изучите алгоритм Дейкстры, который позволяет получить ответ на вопрос «Как выглядит кратчайший путь к X?» для взвешенных графов.
- ✓ Вы узнаете о циклах в графах, для которых алгоритм Дейкстры не работает.

В предыдущей главе вы узнали, как найти путь из точки A в точку B.



Найденный путь не обязательно окажется самым быстрым. Этот путь считается кратчайшим, потому что он состоит из наименьшего количества

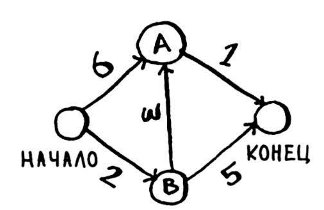
сегментов (три сегмента). Но предположим, с каждым сегментом связывается продолжительность перемещения. И тогда выясняется, что существует и более быстрый путь.



В предыдущей главе рассматривался поиск в ширину. Этот алгоритм находит путь с минимальным количеством сегментов (граф на первом рисунке). А если вы захотите найти самый быстрый путь (второй граф)? Быстрее всего это делается при помощи другого алгоритма, который называется *алгоритмом Дейкстры*.

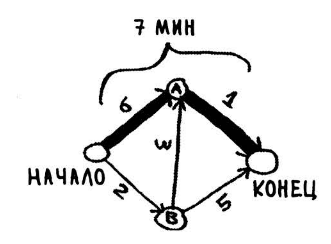
Работа с алгоритмом Дейкстры

Посмотрим, как этот алгоритм работает с графом.



Каждому ребру назначается время перемещения в минутах. Алгоритм Дейкстры используется для поиска пути от начальной точки к конечной за кратчайшее возможное время.

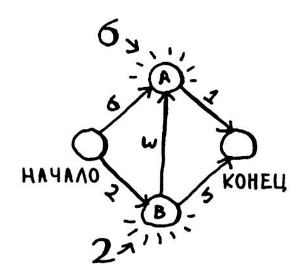
Применив к этому графу поиск в ширину, вы получите следующий кратчайший путь.



Этот путь занимает 7 минут. А может, существует путь, который займет меньше времени? Алгоритм Дейкстры состоит из четырех шагов:

1. Найти узел с наименьшей стоимостью (то есть узел, до которого можно добраться за минимальное время).
2. Обновить стоимости соседей этого узла (вскоре я объясню, что имеется в виду).
3. Повторять, пока это не будет сделано для всех узлов графа.
4. Вычислить итоговый путь.

Шаг 1: найти узел с наименьшей стоимостью. Вы стоите в самом начале и думаете, куда направиться: к узлу A или к узлу B. Сколько времени понадобится, чтобы добраться до каждого из этих узлов?

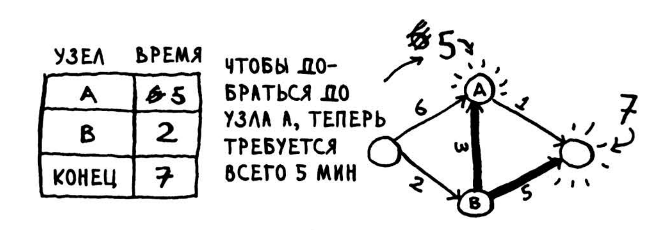


До узла A вы будете добираться 6 минут, а до узла B — 2 минуты. Что касается остальных узлов, мы о них пока ничего не знаем.

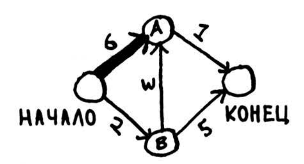
Так как время достижения конечного узла остается неизвестным, мы считаем, что оно бесконечно (вскоре вы увидите почему.) Узел B — ближайший... он находится всего в 2 минутах.

УЗЕЛ	ВРЕМЯ ПЕРЕХОДА К УЗЛУ
A	6
B	2
КОНЕЦ	∞

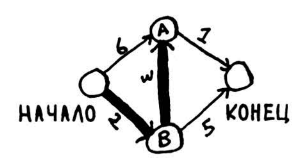
Шаг 2: вычислить, сколько времени потребуется для того, чтобы добраться до всех соседей B *при переходе по ребру из B*.



Ого, да мы обнаружили более короткий путь к узлу A! Раньше для перехода к нему требовалось 6 минут.



А если идти через узел B, то существует путь, который занимает всего 5 минут!



Если вы нашли более короткий путь для соседа B, обновите его стоимость. В данном случае мы нашли:

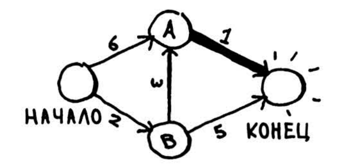
- Более короткий путь к A (сокращение с 6 минут до 5 минут).
- Более короткий путь к конечному узлу (сокращение от бесконечности до 7 минут).

Шаг 3: повторяем!

Снова шаг 1: находим узел, для перехода к которому требуется наименьшее время. С узлом В работа закончена, поэтому наименьшую оценку времени имеет узел А.

УЗЕЛ	ВРЕМЯ	
А	5	←
В	2	
КОНЕЦ	7	

Снова шаг 2: обновляем стоимости соседей А.



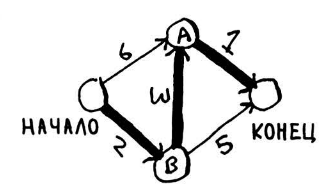
Путь до конечного узла теперь занимает всего 6 минут!

Алгоритм Дейкстры выполнен для каждого узла (выполнять его для конечного узла не нужно). К этому моменту вам известно следующее:

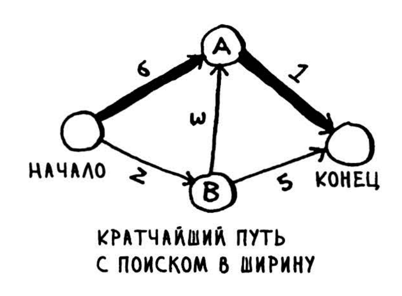
- Чтобы добраться до узла В, нужно 2 минуты.
- Чтобы добраться до узла А, нужно 5 минут.
- Чтобы добраться до конечного узла, нужно 6 минут.

УЗЕЛ	ВРЕМЯ
А	5
В	2
КОНЕЦ	6

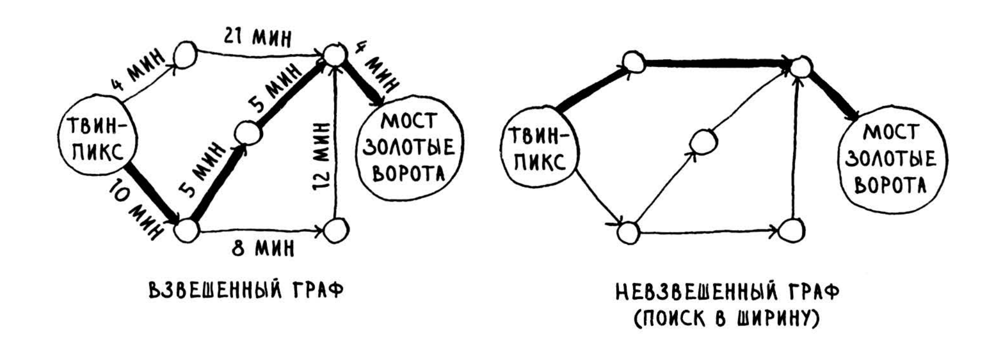
Последний шаг — вычисление итогового пути — откладывается до следующего раздела. А пока я просто покажу, как выглядит итоговый путь.



Алгоритм поиска в ширину не найдет этот путь как кратчайший, потому что он состоит из трех сегментов, а от начального узла до конечного можно добраться всего за два сегмента.



В предыдущей главе мы использовали поиск в ширину для нахождения кратчайшего пути между двумя точками. Тогда под «кратчайшим путем» понимался путь с минимальным количеством сегментов. С другой стороны, в алгоритме Дейкстры каждому сегменту присваивается число (вес), а алгоритм Дейкстры находит путь с наименьшим суммарным весом.



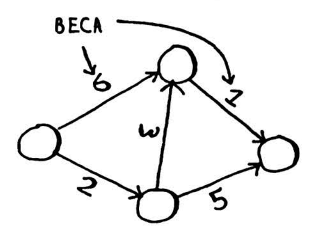
На всякий случай повторим: алгоритм Дейкстры состоит из четырех шагов:

1. Найти узел с наименьшей стоимостью (то есть узел, до которого можно добраться за минимальное время).
2. Проверить, существует ли более дешевый путь к соседям этого узла, и если существует, обновить их стоимости.
3. Повторять, пока это не будет сделано для всех узлов графа.
4. Вычислить итоговый путь (об этом в следующем разделе!).

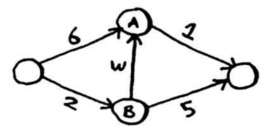
Терминология

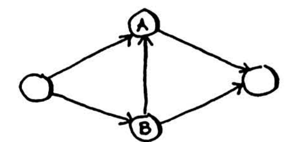
Я хочу привести еще несколько примеров применения алгоритма Дейкстры. Но сначала стоит немного разобраться с терминологией.

Когда вы работаете с алгоритмом Дейкстры, с каждым ребром графа связывается число, называемое *весом*.

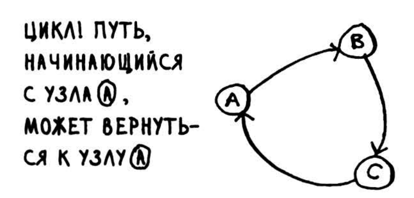


Граф с весами называется *взвешенным графом*. Граф без весов называется *невзвешенным графом*.

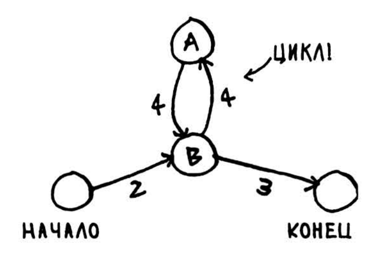




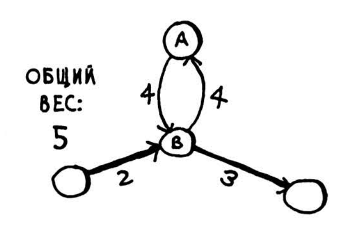
Для вычисления кратчайшего пути в невзвешенном графе используется *поиск в ширину*. Кратчайшие пути во взвешенном графе вычисляются по алгоритму Дейкстры. В графах также могут присутствовать *циклы*:



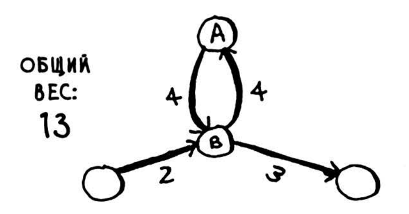
Это означает, что вы можете начать с некоторого узла, перемещаться по графу, а потом снова оказаться в том же узле. Предположим, вы ищете кратчайший путь в графе, содержащем цикл.



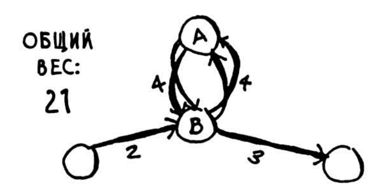
Есть ли смысл в перемещении по циклу? Что ж, вы можете использовать путь без прохождения цикла:



А можете пройти по циклу:



Вы в любом случае оказываетесь в узле А, но цикл добавляет лишний вес. Вы даже можете обойти цикл дважды, если вдруг захотите.

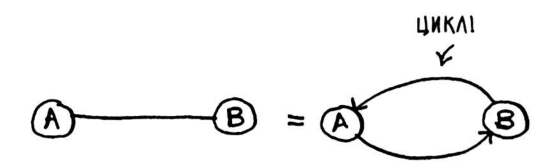


Но каждый раз, когда вы проходите по циклу, вы только увеличиваете суммарный вес на 8. Следовательно, путь с обходом цикла никогда не будет кратчайшим.

Наконец, вы еще не забыли наше обсуждение направленных и ненаправленных графов из главы 6?



Само понятие ненаправленного графа означает, что каждый из двух узлов фактически ведет к другому узлу. А это цикл!



В ненаправленном графе каждое новое ребро добавляет еще один цикл. Алгоритм Дейкстры работает только с *направленными ациклическими графами*, которые нередко обозначаются сокращением DAG (Directed Acyclic Graph).

История одного обмена

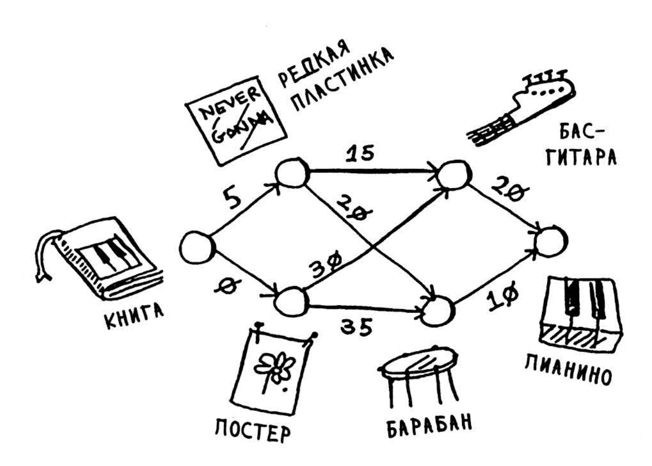


Но довольно терминологии, пора рассмотреть конкретный пример! Это Рама. Он хочет выменять свою книгу по музыке на пианино.

«Я тебе дам за книгу вот этот постер, — говорит Алекс. — Это моя любимая группа Destroyer. Или могу дать за книгу редкую пластинку Рика Эстли и еще $5». — «О, я слышала, что на этой пластинке есть отличные песни, — говорит Эми. — Готова отдать за постер или пластинку мою гитару или ударную установку».

«Всю жизнь мечтал играть на гитаре, — восклицает Бетховен. — Слушай, я отдам тебе свое пианино за любую из вещей Эми».

Прекрасно! Рама с небольшими дополнительными тратами может поменять свою книгу на настоящее пианино. Теперь остается понять, как ему потратить наименьшую сумму на цепочке обменов. Изобразим полученные им предложения в виде графа:



Узлы графа — это предметы, на которые может поменяться Рама. Веса ребер представляют сумму доплаты за обмен. Таким образом, Рама может поменять постер на гитару за $30 или же поменять пластинку на гитару

за $15. Как Раме вычислить путь от книги до пианино, при котором он потратит наименьшую сумму? На помощь приходит алгоритм Дейкстры! Вспомните, что алгоритм Дейкстры состоит из четырех шагов. В этом примере мы выполним все четыре шага, а в конце будет вычислен итоговый путь.

УЗЕЛ	СТОИМОСТЬ
ПЛАСТИНКА	5
ПОСТЕР	∅
ГИТАРА	∞
БАРАБАН	∞
ПИАНИНО	∞

МЫ ЕЩЕ НЕ ДОХОДИЛИ ДО ЭТИХ УЗЛОВ ОТ НАЧАЛЬНОГО

Прежде чем начинать, необходимо немного подготовиться. Постройте таблицу со стоимостями всех узлов. (Стоимость узла определяет затраты на его достижение.)

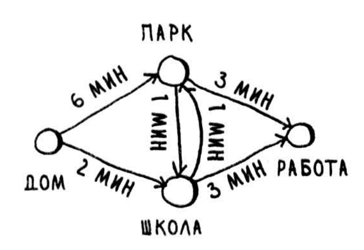
Таблица будет обновляться по мере работы алгоритма. Для вычисления итогового пути в таблицу также необходимо добавить столбец «родитель».

УЗЕЛ	РОДИТЕЛЬ
ПЛАСТИНКА	КНИГА
ПОСТЕР	КНИГА
ГИТАРА	—
БАРАБАН	—
ПИАНИНО	—

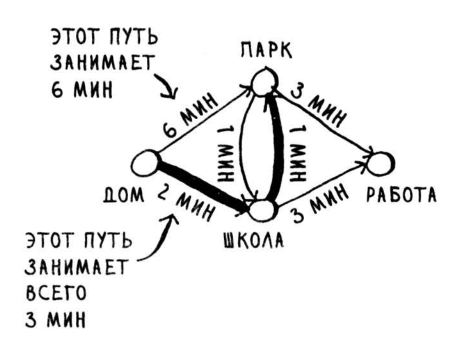
Вскоре я покажу, как работает этот столбец. А пока просто запустим алгоритм.

Шаг 1: найти узел с наименьшей стоимостью. В данном случае самый дешевый вариант обмена с доплатой $0 — это постер. Возможно ли получить постер с меньшими затратами? Это очень важный момент, хорошенько подумайте над ним. Удастся ли вам найти серию обменов, при которой Рама

получит постер менее чем за $0? Продолжайте читать, когда будете готовы ответить на вопрос. Правильный ответ: нет, не удастся. *Так как постер является узлом с наименьшей стоимостью, до которого может добраться Рама*, снизить его стоимость невозможно. На происходящее можно взглянуть иначе: предположим, вы едете из дома на работу.



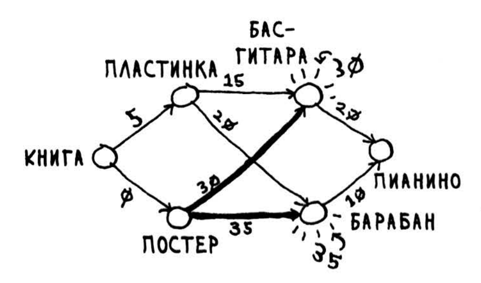
Если вы выберете путь к школе, это займет 2 минуты. Если вы выберете путь к парку, это займет 6 минут. Существует ли путь, при котором вы выбираете путь к парку и оказываетесь в школе менее чем за 2 минуты? Это невозможно, потому что только для того, чтобы попасть в парк, потребуется более 2 минут. С другой стороны, можно ли найти более быстрый путь в парк? Да, можно.



В этом заключается ключевая идея алгоритма Дейкстры: *в графе ищется путь с наименьшей стоимостью. Пути к этому узлу с меньшими затратами не существует!*

Возвращаемся к музыкальному примеру. Вариант с постером обладает наименьшей стоимостью.

Шаг 2: Вычислить, сколько времени потребуется для того, чтобы добраться до всех его соседей (стоимость).



РОДИТЕЛЬ	УЗЕЛ	СТОИМОСТЬ
КНИГА		5
КНИГА	ПОСТЕР	0
ПОСТЕР		~~∞~~ 30
ПОСТЕР	БАРАБАН	~~∞~~ 35
—	ПИАНИНО	∞

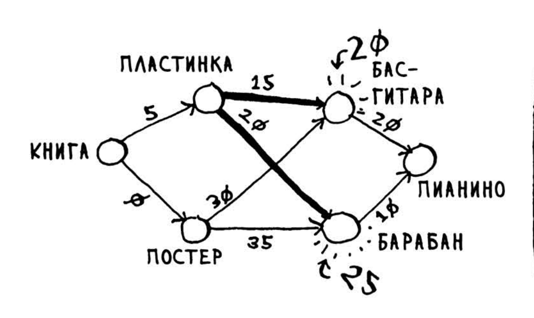
Стоимости бас-гитары и барабана заносятся в таблицу. Они были заданы при переходе через узел постера, поэтому постер указывается как их родитель. А это означает, что для того, чтобы добраться до бас-гитары, вы проходите по ребру от постера; то же самое происходит с барабаном.

К ЭТИМ УЗЛАМ ПЕРЕХОДИМ ОТ УЗЛА «ПОСТЕР» (строки ГИТАРА и БАРАБАН)

РОДИТЕЛЬ	УЗЕЛ	СТОИМОСТЬ
КНИГА	ПЛАСТИНКА	5
КНИГА	ПОСТЕР	0
ПОСТЕР	ГИТАРА	30
ПОСТЕР	БАРАБАН	35
—	ПИАНИНО	∞

Снова шаг 1: пластинка — следующий по стоимости узел ($5).

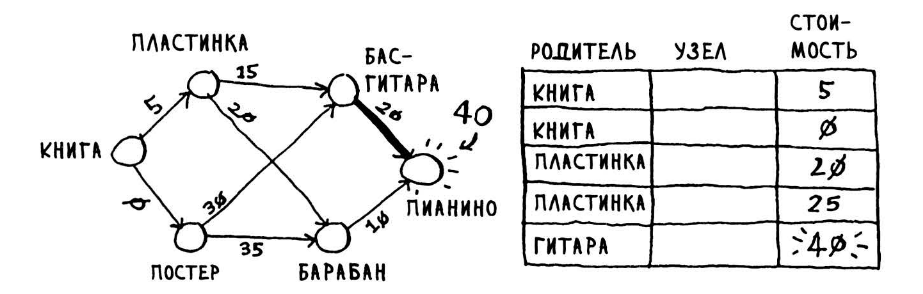
Снова шаг 2: обновляются значения всех его соседей.



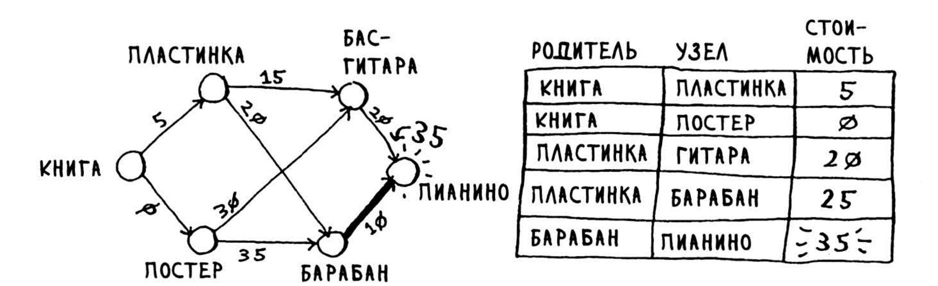
РОДИТЕЛЬ	УЗЕЛ	СТОИМОСТЬ
КНИГА	ПЛАСТИНКА	5
КНИГА	ПОСТЕР	0
ПЛАСТИНКА	ГИТАРА	~~30~~ 20
ПЛАСТИНКА	БАРАБАН	~~35~~ 25
—	ПИАНИНО	∞

Смотрите, стоимости барабана и гитары обновились! Это означает, что к барабану и гитаре дешевле перейти через ребро, идущее от пластинки. Соответственно, пластинка назначается новым родителем обоих инструментов.

Следующий по стоимости узел — бас-гитара. Обновите данные его соседей.



Хорошо, мы наконец-то вычислили стоимость для пианино при условии обмена гитары на пианино. Соответственно, гитара назначается родителем. Наконец, задается стоимость последнего узла — барабана.

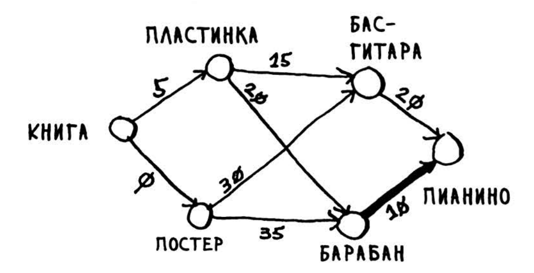


Оказывается, Рама может получить пианино еще дешевле, поменяв ударную установку на пианино. Таким образом, *самая дешевая цепочка обменов обойдется Раме в $35.*

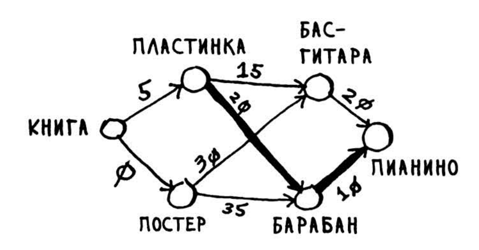
Теперь, как я и обещал, необходимо вычислить итоговый путь. К этому моменту вы уже знаете, что кратчайший путь обойдется в $35, но как этот путь определить? Для начала возьмем родителя узла «пианино».

РОДИТЕЛЬ	УЗЕЛ
КНИГА	ПЛАСТИНКА
КНИГА	ПОСТЕР
ПЛАСТИНКА	ГИТАРА
ПЛАСТИНКА	БАРАБАН
→ БАРАБАН	ПИАНИНО

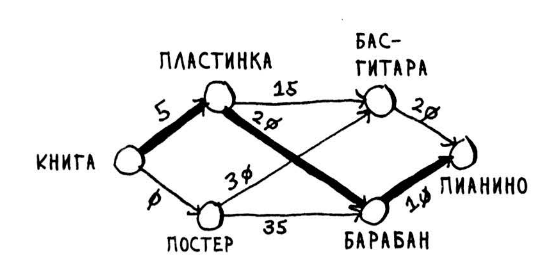
В качестве родителя узла «пианино» указан узел «барабан».



А в качестве родителя узла «барабан» указан узел «пластинка».



Следовательно, Рама обменивает пластинку на барабан. И конечно, в самом начале он меняет книгу на пластинку. Проходя по родительским узлам в обратном направлении, мы получаем полный путь.

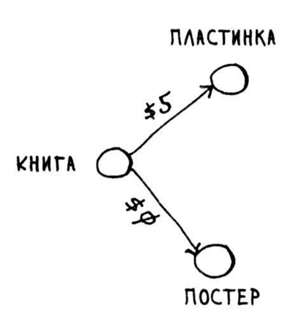


Серия обменов, которую должен сделать Рама, выглядит так:



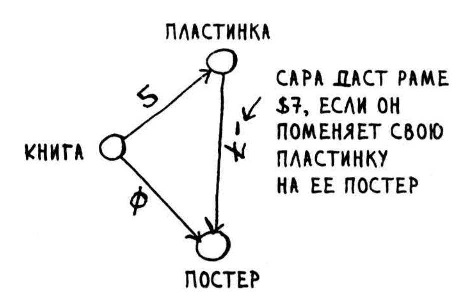
До сих пор я использовал термин «кратчайший путь» более или менее буквально, понимая под ним вычисление кратчайшего пути между двумя точками или двумя людьми. Надеюсь, этот пример показал, что кратчайший путь далеко не всегда связывается с физическим расстоянием: он может быть направлен на минимизацию какой-либо характеристики. В нашем примере Рама хотел свести к минимуму свои затраты при обмене. Спасибо Дейкстре!

Ребра с отрицательным весом

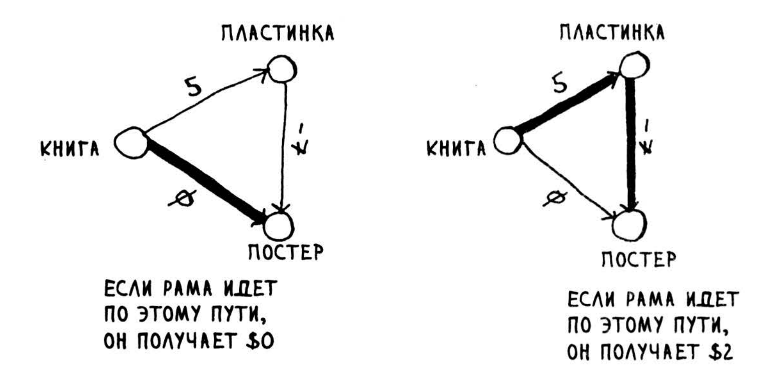


В предыдущем примере Алекс предложил в обмен на книгу один из двух предметов.

Предположим, Сара предложила обменять пластинку на постер и при этом она еще и *даст Раме* $7. Рама ничего не тратит при этом обмене, вместо этого он получит $7. Как изобразить это предложение на графе?

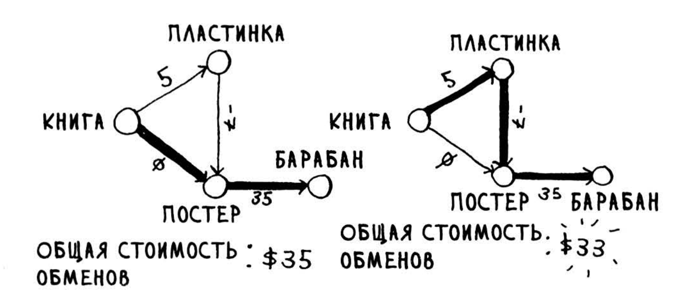


Ребро, ведущее от пластинки к постеру, имеет отрицательный вес! Если Рама пойдет на этот обмен, он получит $7. Теперь к постеру можно добраться двумя способами.



А значит, во втором обмене появляется смысл — Рама получает $2!

Теперь, если вы помните, Рама может обменять постер на барабан. И здесь возможны два пути.



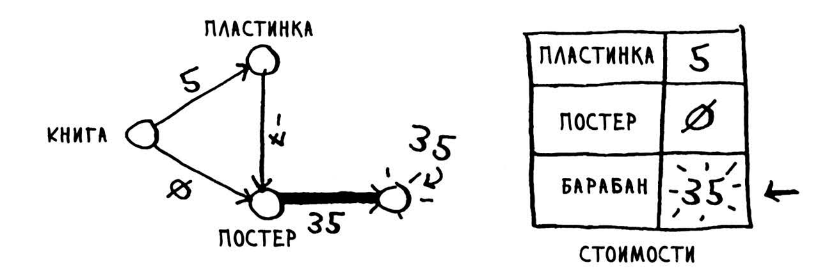
Второй путь обойдется на $2 дешевле, поэтому нужно выбрать этот путь, верно?

И знаете что? Если применить алгоритм Дейкстры к этому графу, Рама выберет неверный путь. Он пойдет по более длинному пути. Алгоритм Дейкстры не может использоваться при наличии ребер, имеющих отрицательный вес. Такие ребра нарушают работу алгоритма. Посмотрим, что произойдет, если попытаться применить алгоритм Дейкстры к этому графу. Все начинается с построения таблицы стоимостей.



Теперь найдем узел с наименьшей стоимостью и обновим стоимости его соседей. В этом случае постер оказывается узлом с наименьшей стоимостью. Итак, в соответствии с алгоритмом Дейкстры, к постеру *невозможно перей-*

ти более дешевым способом, чем с оплатой $0 (а вы знаете, что это неверно!) Как бы то ни было, обновим стоимости его соседей.

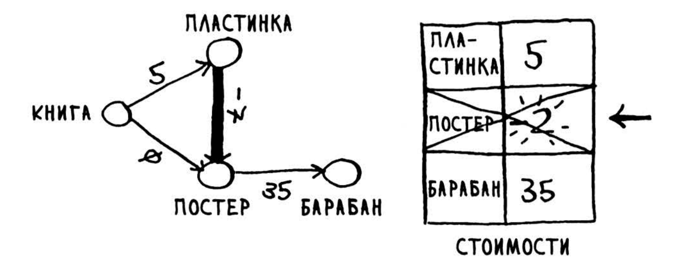


Получается, что теперь стоимость барабана составляет $35.

Перейдем к следующему по стоимости узлу, который еще не был обработан.



Обновим стоимости его соседей.



Узел «постер» уже был обработан, однако вы обновляете его стоимость. Это очень тревожный признак — обработка узла означает, что к нему не-

возможно добраться с меньшими затратами. Но вы только что нашли более дешевый путь к постеру! У барабана соседей нет, поэтому работа алгоритма завершена. Ниже приведены итоговые стоимости.

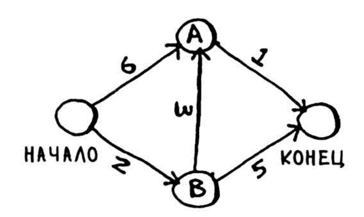
ПЛАСТИНКА	5
ПОСТЕР	-2
БАРАБАН	35

ИТОГОВЫЕ СТОИМОСТИ

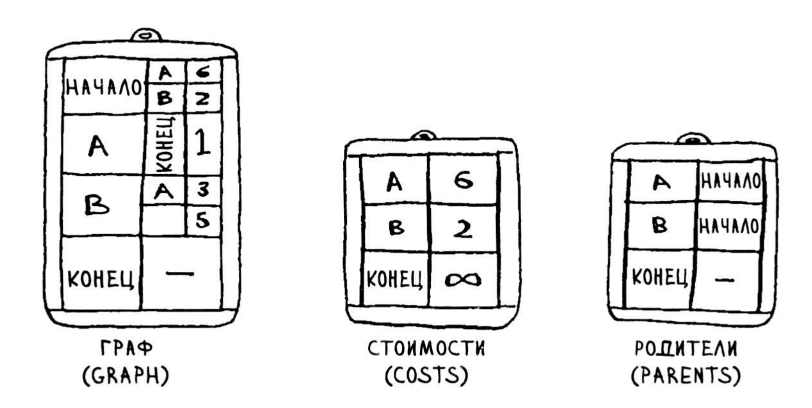
Чтобы добраться до барабанов, Раме потребовалось $35. Вы знаете, что существует путь, который стоит всего $33, но алгоритм Дейкстры его не находит. Алгоритм Дейкстры предположил, что, поскольку вы обрабатываете узел «постер», к этому узлу невозможно добраться быстрее. Это предположение работает только в том случае, если ребер с отрицательным весом не существует. Следовательно, *использование алгоритма Дейкстры с графом, содержащим ребра с отрицательным весом, невозможно*. Если вы хотите найти кратчайший путь в графе, содержащем ребра с отрицательным весом, для этого существует специальный алгоритм, называемый *алгоритмом Беллмана—Форда*. Рассмотрение этого алгоритма выходит за рамки этой книги, но вы сможете найти хорошие описания в Интернете.

Реализация

Посмотрим, как алгоритм Дейкстры реализуется в программном коде. Ниже изображен граф, который будет использоваться в этом примере.



Для реализации этого примера понадобятся три хеш-таблицы.



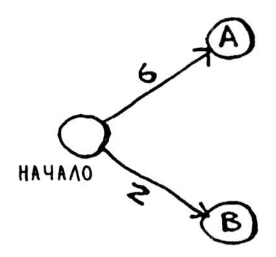
Хеш-таблицы стоимостей и родителей будут обновляться по ходу работы алгоритма. Сначала необходимо реализовать граф. Как и в главе 6, для этого будет использована хеш-таблица:

```
graph = {}
```

В предыдущей главе все соседи узла были сохранены в хеш-таблице:

```
graph["you"] = ["alice", "bob", "claire"]
```

Но на этот раз необходимо сохранить как соседей, так и стоимость перехода к соседу. Предположим, у начального узла есть два соседа, A и B.



Как представить веса этих ребер? Почему бы не воспользоваться другой хеш-таблицей?

```
graph["start"] = {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2
```



Итак, `graph["start"]` является хеш-таблицей. Для получения всех соседей начального узла можно воспользоваться следующим выражением:

```
>>> print graph["start"].keys()
["a", "b"]
```

Одно ребро ведет из начального узла в A, а другое — из начального узла в B. А если вы захотите узнать веса этих ребер?

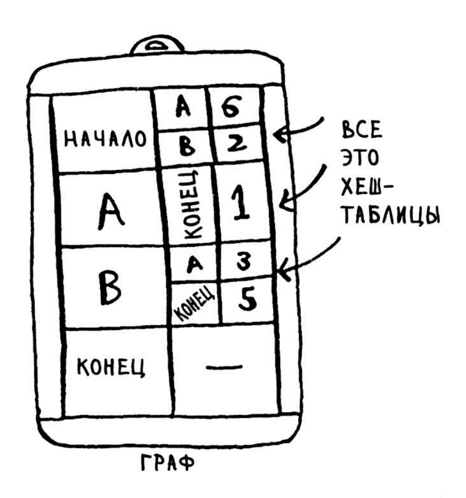
```
>>> print graph["start"]["a"]
2
>>> print graph["start"]["b"]
6
```

Включим в граф остальные узлы и их соседей:

```
graph["a"] = {}
graph["a"]["fin"] = 1
graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5
graph["fin"] = {}
```

У конечного узла нет соседей

Полная хеш-таблица графа выглядит так:





Также понадобится хеш-таблица для хранения стоимостей всех узлов.

Стоимость узла определяет, сколько времени потребуется для перехода к этому узлу от начального узла. Вы знаете, что переход от начального узла к узлу B занимает 2 минуты. Вы знаете, что для перехода к узлу A требуется 6 минут (хотя, возможно, вы найдете более быстрый путь). Вы не знаете, сколько времени потребуется для достижения конечного узла. Если стоимость еще неизвестна, она считается бесконечной. Можно ли представить *бесконечность* в Python? Оказывается, можно:

```
infinity = float("inf")
```

Код создания таблицы стоимостей `costs`:

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

Для родителей также создается отдельная таблица:



Код создания хеш-таблицы родителей:

```
parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["in"] = None
```

Наконец, вам нужен массив для отслеживания всех уже обработанных узлов, так как один узел не должен обрабатываться многократно:

```
processed = []
```

На этом подготовка завершается. Теперь обратимся к алгоритму.



Сначала я приведу код, а потом мы разберем его более подробно.

```
node = find_lowest_cost_node(costs)      # Найти узел с наименьшей стоимостью среди необработанных
while node is not None:                  # Если обработаны все узлы, цикл while завершен
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():           # Перебрать всех соседей текущего узла
        new_cost = cost + neighbors[n]   # Если к соседу можно быстрее
        if costs[n] > new_cost:          # добраться через текущий узел...
            costs[n] = new_cost          # ...обновить стоимость для этого узла
            parents[n] = node            # Этот узел становится новым родителем для соседа
    processed.append(node)               # Узел помечается как обработанный
    node = find_lowest_cost_node(costs)  # Найти следующий узел для обработки и повторить цикл
```

Так выглядит алгоритм Дейкстры на языке Python! Код функции будет приведен далее, а пока рассмотрим пример использования алгоритма в действии.

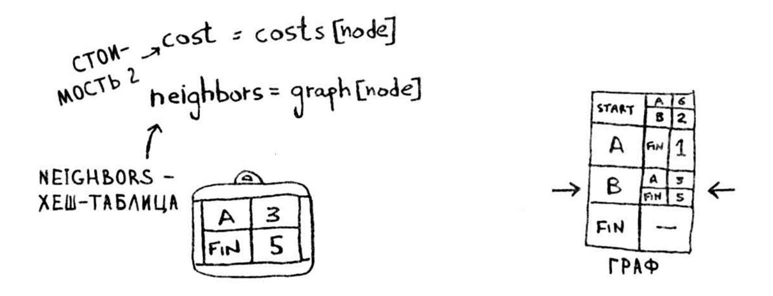
Найти узел с наименьшей стоимостью.

ЭТО УЗЕЛ «B» → node = find_lowest_cost_node(costs)

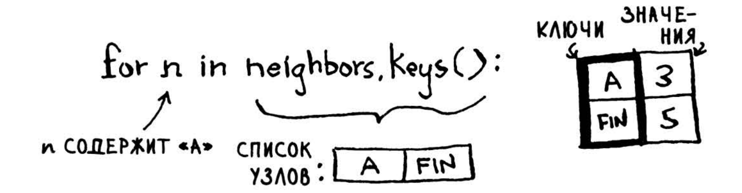
A	6
B	2
КОНЕЦ	∞

СТОИМОСТИ

Получить стоимость и соседей этого узла.



Перебрать соседей.



У каждого узла имеется стоимость, которая определяет, сколько времени потребуется для достижения этого узла от начала. Здесь мы вычисляем, сколько времени потребуется для достижения узла A по пути Начало > Узел B > Узел A (вместо Начало > Узел A).

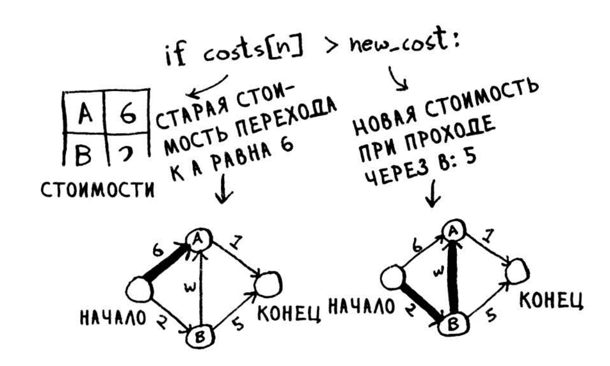
```
new_cost = cost + neighbors[n]
```

СТОИМОСТЬ «B», т. е. 2

РАССТОЯНИЕ ОТ B ДО A: 3

new_cost = 2+3 = 5

Сравним эти стоимости.



Мы нашли более короткий путь к узлу A! Обновим стоимость.

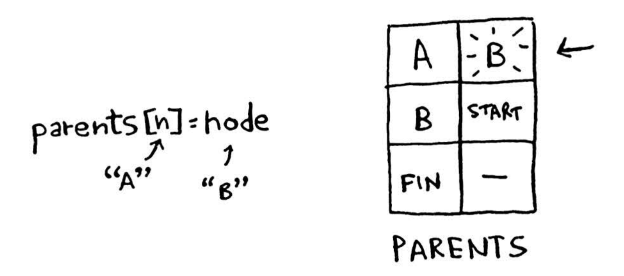
```
costs[n] = new_cost
```

"A" — 5

A	~~6~~ 5
B	2
FIN	∞

COSTS

Новый путь проходит через узел B, поэтому B назначается новым родителем.



Мы снова вернулись к началу цикла. Следующим соседом в цикле `for` является конечный узел.

for n in neighbors.keys():

n содержит «FIN»

A | FIN

Сколько времени потребуется для достижения конечного узла, если идти через узел B?

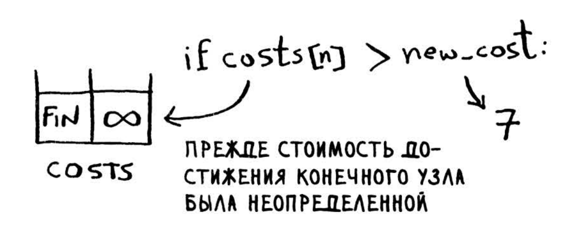
new_cost = cost + neighbors[n]

2

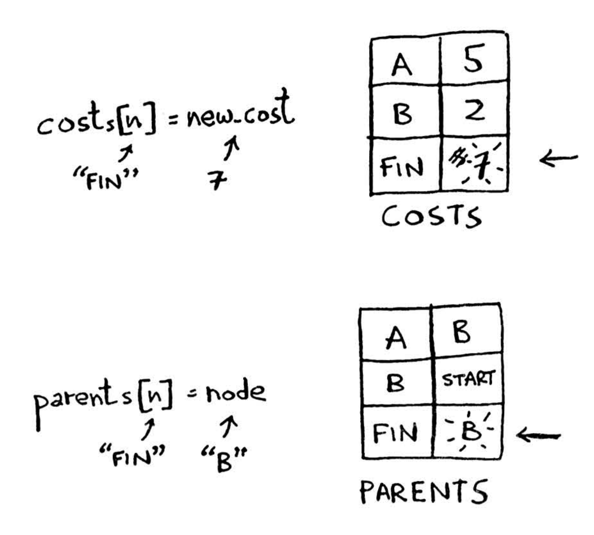
РАССТОЯНИЕ ОТ B ДО КОНЦА: 5

2 + 5 = 7

Потребуется 7 минут. Предыдущая стоимость была бесконечной, а 7 минут определенно меньше бесконечности.



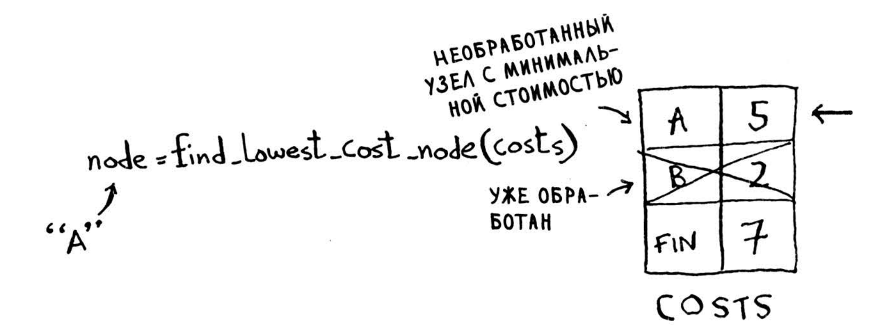
Конечному узлу назначается новая стоимость и новый родитель.



Порядок, мы обновили стоимости всех соседей узла B. Узел помечается как обработанный.

processed.append(node)
"B"
ОБРАБОТАННЫЕ УЗЛЫ: B

Найти следующий узел для обработки.



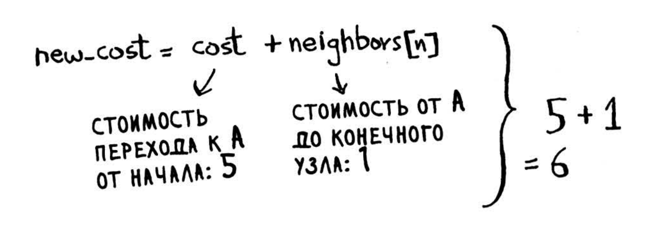
Получить стоимость и соседей узла А.

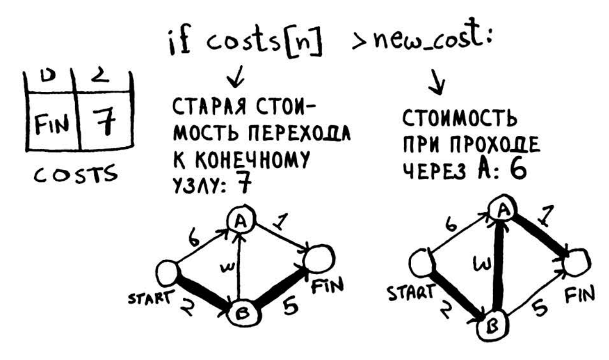
```
cost = costs[node]
5
neighbors = graph[node]
FIN | 1
```

У узла А всего один сосед: конечный узел.

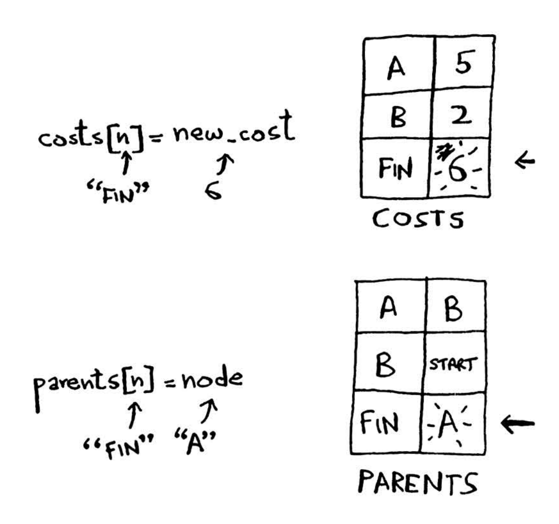
```
for n in neighbors.keys():
"FIN"          FIN
```

Время достижения конечного узла составляет 7 минут. Сколько времени потребуется для достижения конечного узла, если идти через узел А?





Через узел A можно добраться быстрее! Обновим стоимость и родителя.



После того как все узлы будут обработаны, алгоритм завершается. Надеюсь, этот пошаговый разбор помог вам чуть лучше понять алгоритм. С функцией `find_lowest_cost_node` узел с наименьшей стоимостью находится проще простого. Код выглядит так:

```
def ind_lowest_cost_node(costs):
    lowest_cost = loat("inf")
    lowest_cost_node = None
    for node in costs:  <······  Перебрать все узлы
        cost = costs[node]
        if cost < lowest_cost and node not in processed: <······
            lowest_cost = cost  <······
            lowest_cost_node = node
    return lowest_cost_node
```

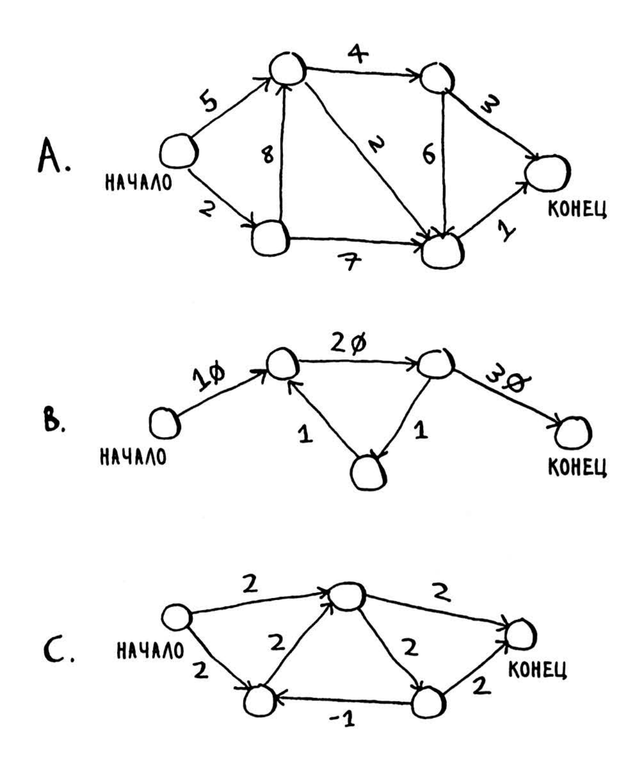
Перебрать все узлы

Если это узел с наименьшей стоимостью из уже виденных и он еще не был обработан...

...он назначается новым узлом с наименьшей стоимостью

Упражнения

7.1 Каков вес кратчайшего пути от начала до конца в каждом из следующих графов?



Шпаргалка

- Поиск в ширину вычисляет кратчайший путь в невзвешенном графе.
- Алгоритм Дейкстры вычисляет кратчайший путь во взвешенном графе.
- Алгоритм Дейкстры работает только в том случае, если все веса положительны.
- При наличии отрицательных весов используйте алгоритм Беллмана—Форда.

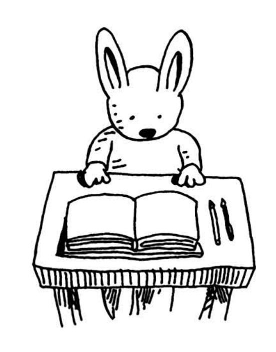
8 Жадные алгоритмы



В этой главе

- ✓ Вы узнаете, как браться за невозможные задачи, не имеющие быстрого алгоритмического решения (NP-полные задачи).
- ✓ Вы научитесь узнавать такие задачи и не терять время на поиски быстрого алгоритма (которого все равно нет).
- ✓ Вы познакомитесь с приближенными алгоритмами, которые могут использоваться для быстрого нахождения приближенного решения NP-полных задач.
- ✓ Вы узнаете о жадной стратегии — очень простой стратегии решения задач.

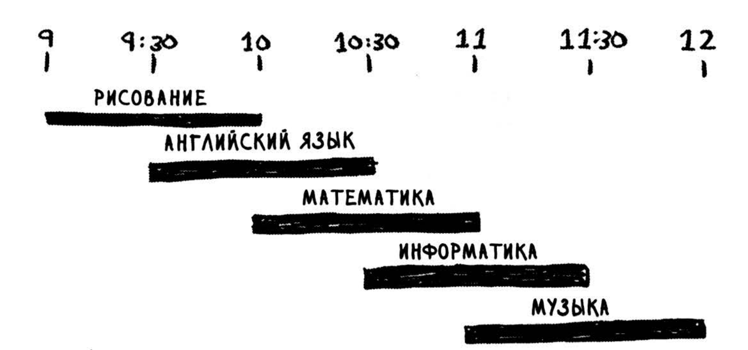
Задача составления расписания



Допустим, имеется учебный класс, в котором нужно провести как можно больше уроков. Вы получаете список уроков.

ПРЕДМЕТ	С	ДО
РИС.	9:00	10:00
АНГЛ.	9:30	10:30
МАТ-КА	10:00	11:00
ИНФ-КА	10:30	11:30
МУЗЫКА	11:00	12:00

Провести в классе *все* уроки не получится, потому что некоторые из них перекрываются по времени.



Требуется провести в классе как можно больше уроков. Как отобрать уроки, чтобы полученный набор оказался самым большим из возможных?

Вроде бы сложная задача, верно? На самом деле алгоритм оказывается на удивление простым. Вот как он работает:

1. Выбрать урок, завершающийся раньше всех. Это первый урок, который будет проведен в классе.

2. Затем выбирается урок, начинающийся после завершения первого урока. И снова следует выбрать урок, который завершается раньше всех остальных. Он становится вторым уроком в расписании.

Продолжайте действовать по тому же принципу — и вы получите ответ! Давайте попробуем. Рисование заканчивается раньше всех уроков (в 10:00), поэтому мы выбираем именно его.

РИС.	9:00	10:00	✓
АНГЛ.	9:30	10:30	
МАТ-КА	10:00	11:00	
ИНФ-КА	10:30	11:30	
МУЗЫКА	11:00	12:00	

Теперь нужно найти следующий урок, который начинается после 10:00 и завершается раньше остальных.

РИС.	9:00	10:00	✓
АНГЛ.	9:30	10:30	✗
МАТ-КА	10:00	11:00	✓
ИНФ-КА	10:30	11:30	
МУЗЫКА	11:00	12:00	

Английский язык отпадает — он перекрывается с рисованием, но математика подходит. Наконец, информатика перекрывается с математикой, но музыка подходит.

РИС.	9:00	10:00	✓
АНГЛ.	9:30	10:30	✗
МАТ-КА	10:00	11:00	✓
ИНФ-КА	10:30	11:30	✗
МУЗЫКА	11:00	12:00	✓

Итак, эти три урока должны проводиться в классе.



Я очень часто слышу, что этот алгоритм подозрительно прост. Он слишком очевиден, а значит, должен быть неправильным. Но в этом и заключается красота жадных алгоритмов: они просты! Жадный алгоритм прост: на каждом шаге он выбирает оптимальный вариант. В нашем примере при выборе урока выбирается тот урок, который завершается раньше других. В технической терминологии: на каждом шаге выбирается *локально-оптимальное решение*, а в итоге вы получаете глобально-оптимальное решение. Хотите верьте, хотите нет, но этот простой алгоритм успешно находит оптимальное решение задачи составления расписания!

Конечно, жадные алгоритмы работают не всегда. Но они так просто реализуются! Рассмотрим другой пример.

Задача о рюкзаке



Представьте, что вы жадный воришка. Вы забрались в магазин с рюкзаком, и перед вами множество товаров, которые вы можете украсть. Однако емкость рюкзака не бесконечна: он выдержит не более 35 фунтов.



Требуется подобрать набор товаров максимальной стоимости, которые можно сложить в рюкзак. Какой алгоритм вы будете использовать?

И снова жадная стратегия выглядит очень просто:

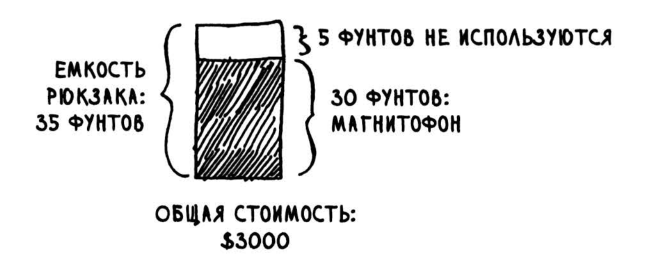
1. Выбрать самый дорогой предмет, который поместится в рюкзаке.

2. Выбрать следующий по стоимости предмет, который поместится в рюкзаке... И так далее.

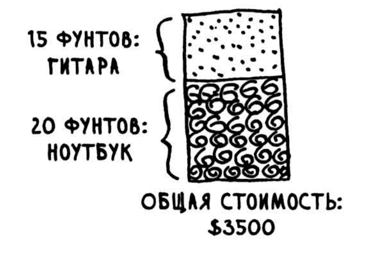
Вот только на этот раз она не работает! Предположим, есть три предмета.



В рюкзаке поместятся товары общим весом не более 35 фунтов. Самый дорогой товар — магнитофон, вы выбираете его. Теперь ни для чего другого места уже не осталось.



Вы набрали товаров на $3000. Погодите-ка! Если бы вместо магнитофона вы выбрали ноутбук и гитару, то стоимость добычи составила бы $3500!



Очевидно, жадная стратегия не дает оптимального решения. Впрочем, результат не так уж далек от оптимума. В следующей главе я расскажу, как вычислить правильное решение. Но вор, забравшийся в магазин, вряд ли станет стремиться к идеалу. «Достаточно хорошего» решения должно хватить.

Второй пример приводит нас к следующему выводу: иногда идеальное — враг хорошего. В некоторых случаях достаточно алгоритма, способного решить задачу достаточно хорошо. И в таких областях жадные алгоритмы работают просто отлично, потому что они просто реализуются, а полученное решение обычно близко к оптимуму.

Упражнения

8.1 Вы работаете в фирме по производству мебели и поставляете мебель по всей стране. Коробки с мебелью размещаются в грузовике. Все коробки имеют разный размер, и вы стараетесь наиболее эффективно использовать доступное пространство. Как выбрать коробки для того, чтобы загрузка имела максимальную эффективность? Предложите жадную стратегию. Будет ли полученное решение оптимальным?

8.2 Вы едете в Европу, и у вас есть семь дней на знакомство с достопримечательностями. Вы присваиваете каждой достопримечательности стоимость в баллах (насколько вы хотите ее увидеть) и оцениваете продолжительность поездки. Как обеспечить максимальную стоимость (увидеть все самое важное) во время поездки? Предложите жадную стратегию. Будет ли полученное решение оптимальным?

Рассмотрим еще один пример, в котором без жадных алгоритмов практически не обойтись.

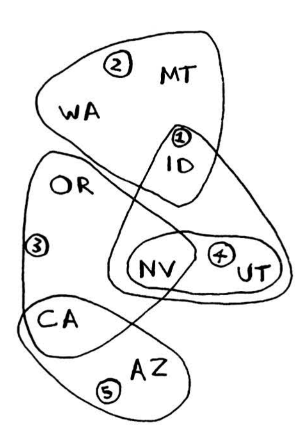
Задача о покрытии множества

Вы открываете собственную авторскую программу на радио и хотите, чтобы вас слушали во всех 50 штатах. Нужно решить, на каких радиостанциях должна транслироваться ваша передача. Каждая станция стоит денег, поэтому количество станций необходимо свести к минимуму. Имеется список станций.

РАДИО-СТАНЦИЯ	ДОСТУПНА В ШТАТАХ
KONE	ID, NV, UT
KTWO	WA, ID, MT
KTHREE	OR, NV, CA
KFOUR	NV, UT
KFIVE	CA, AZ

... и т. д. ...

Каждая станция покрывает определенный набор штатов, эти наборы перекрываются.



Как найти минимальный набор станций, который бы покрывал все 50 штатов? Вроде бы простая задача, верно? Оказывается, она чрезвычайно сложна. Вот как это делается:

1. Составить список всех возможных подмножеств станций — так называемое *степенное множество*. В нем содержатся 2^*n* возможных подмножеств.

2. Из этого списка выбирается множество с наименьшим набором станций, покрывающих все 50 штатов.



Проблема в том, что вычисление всех возможных подмножеств станций займет слишком много времени. Для *n* станций оно потребует времени $O(2\hat{}n)$. Если станций немного, скажем от 5 до 10, — это допустимо. Но подумайте, что произойдет во всех рассмотренных примерах при большом количестве элементов. Предположим, вы можете вычислять по 10 подмножеств в секунду.

Не существует алгоритма, который будет вычислять подмножества с приемлемой скоростью! Что же делать?

КОЛИЧЕСТВО СТАНЦИЙ	НЕОБХОДИМОЕ ВРЕМЯ
5	3.2 с
10	102.4 с
32	13.6 года
100	4×10^{21} года

Приближенные алгоритмы

На помощь приходят жадные алгоритмы! Вот как выглядит жадный алгоритм, который выдает результат, достаточно близкий к оптимуму:

1. Выбрать станцию, покрывающую наибольшее количество штатов, еще не входящих в покрытие. Если станция будет покрывать некоторые штаты, уже входящие в покрытие, это нормально.

2. Повторять, пока остаются штаты, не входящие в покрытие.

Этот алгоритм является *приближенным*. Когда вычисление точного решения занимает слишком много времени, применяется приближенный алгоритм. Эффективность приближенного алгоритма оценивается по:

- быстроте;
- близости полученного решения к оптимальному.

Жадные алгоритмы хороши не только тем, что они обычно легко формулируются, но и тем, что простота обычно оборачивается быстротой выполнения. В данном случае жадный алгоритм выполняется за время $O(n\hat{}2)$, где n — количество радиостанций.

А теперь посмотрим, как эта задача выглядит в программном коде.

Подготовительный код

В этом примере для простоты будет использоваться небольшое подмножество штатов и станций.

Сначала составьте список штатов:

```
states_needed = set(["mt", "wa", "or", "id", "nv", "ut",
"ca",  "az"])
```

Переданный массив преобразуется в множество

В этой реализации я использовал множество. Эта структура данных похожа на список, но каждый элемент может встречаться в множестве не более одного раза. *Множества не содержат дубликатов*. Предположим, имеется следующий список:

```
>>> arr = [1, 2, 2, 3, 3, 3]
```

Этот список преобразуется в множество:

```
>>> set(arr)
set([1, 2, 3])
```

Значения 1, 2 и 3 встречаются в списке по одному разу.

[1,2,2,3,3,3] → ПРЕОБРАЗУЕТСЯ ВО МНОЖЕСТВО → (1,2,3)

МНОЖЕСТВО

Также понадобится список станций, из которого будет выбираться покрытие. Я решил воспользоваться хешем:

```
stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])
```

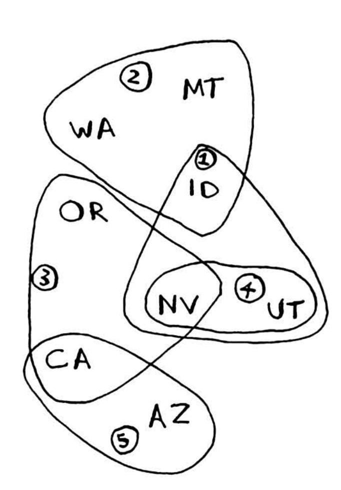
Ключи — названия станций, а значения — сокращенные обозначения штатов, входящих в зону охвата. Таким образом, в данном примере станция kone вещает в штатах Айдахо (*id*), Невада (*nv*) и Юта (*ut*). Все значения являются множествами. Как вы вскоре увидите, хранение данных во множествах упрощает работу.

Наконец, нам понадобится структура данных для хранения итогового набора станций:

```
final_stations = set()
```

Вычисление ответа

Теперь необходимо вычислить набор используемых станций. Взгляните на диаграмму и попробуйте предсказать, какие станции следует использовать.



Учтите, что правильных решений может быть несколько. Вы перебираете все станции и выбираете ту, которая обслуживает больше всего штатов, не входящих в текущее покрытие. Будем называть ее `best_station`:

```
best_station = None
states_covered = set()
for station, states_for_station in stations.items():
```

Множество `states_covered` содержит все штаты, обслуживаемые этой станцией, которые еще не входят в текущее покрытие. Цикл `for` перебирает все станции и находит среди них наилучшую. Рассмотрим тело цикла `for`:

```
covered = states_needed & states_for_station
if len(covered) > len(states_covered)
  best_station = station
  states_covered = covered
```

Новый синтаксис! Эта операция называется "пересечением множеств"

В коде встречается необычная строка:

```
covered = states_needed & states_for_station
```

Что здесь происходит?

Множества

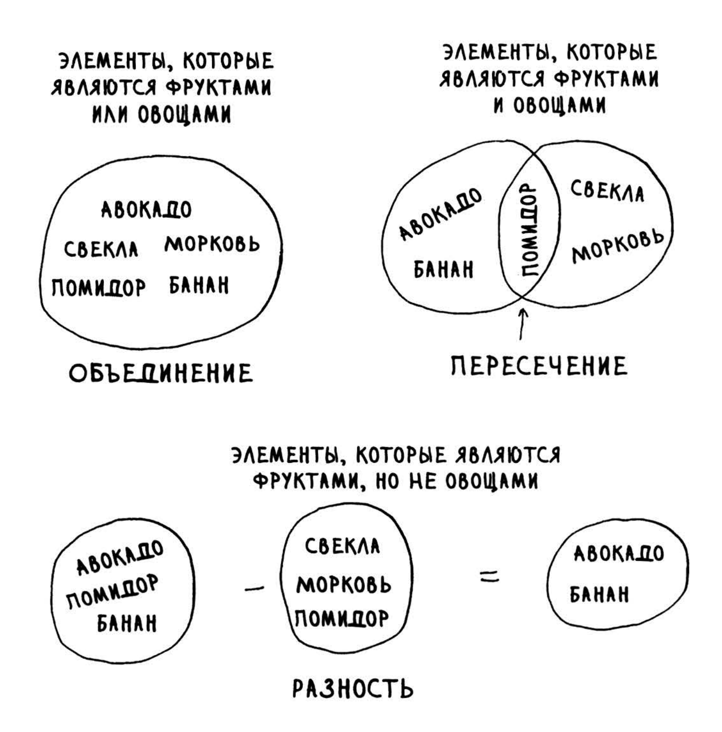
Допустим, имеется множество с названиями фруктов.



Также имеется множество с названиями овощей.



С двумя множествами можно выполнить ряд интересных операций.



- Объединение множеств означает слияние элементов обоих множеств.
- Под операцией пересечения множеств понимается поиск элементов, входящих в оба множества (в данном случае — только помидор).
- Под разностью множеств понимается исключение из одного множества элементов, присутствующих в другом множестве.

Пример:

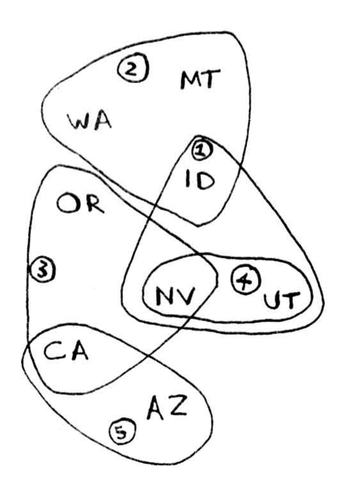
```
>>> fruits = set(["avocado", "tomato", "banana"])
>>> vegetables = set(["beets", "carrots", "tomato"])
>>> fruits | vegetables     <·················   Объединение множеств
set(["avocado", "beets", "carrots", "tomato", "banana"])
>>> fruits & vegetables     <·················   Пересечение множеств
set(["tomato"])
>>> fruits - vegetables     <·················   Разность множеств
set(["avocado", "banana"])
>>> vegetables - fruits     <·················   Как вы думаете, как будет выглядеть результат?
```

Еще раз напомню основные моменты:

- множества похожи на списки, но множества не содержат дубликатов;
- с множествами можно выполнять различные интересные операции — вычислять их объединение, пересечение и разность.

Вернемся к коду

Продолжим рассматривать исходный пример.



Пересечение множеств:

```
covered = states_needed & states_for_station
```

Множество `covered` содержит штаты, присутствующие как в `states_needed`, так и в `states_for_station`. Таким образом, `covered` — множество штатов, не входящих в покрытие, которые покрываются текущей станцией! Затем мы проверяем, покрывает ли эта станция больше штатов, чем текущая станция `best_station`:

```
if len(covered) > len(states_covered):
  best_station = station
  states_covered = covered
```

Если условие выполняется, то станция сохраняется в `best_station`. Наконец, после завершения цикла `best_station` добавляется в итоговый список станций:

```
final_stations.add(best_station)
```

Также необходимо обновить содержимое `states_needed`. Те штаты, которые входят в зону покрытия станции, больше не нужны:

```
states_needed -= states_covered
```

Цикл продолжается, пока множество `states_needed` не станет пустым. Полный код цикла `for` выглядит так:

```
while states_needed:
  best_station = None
  states_covered = set()
  for station, states in stations.items():
    covered = states_needed & states
    if len(covered) > len(states_covered):
      best_station = station
      states_covered = covered

states_needed -= states_covered
final_stations.add(best_station)
```

Остается вывести содержимое `final_stations`:

```
>>> print final_stations
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

Этот результат совпадает с вашими ожиданиями? Вместо станций 1, 2, 3 и 5 можно было выбрать станции 2, 3, 4 и 5. Сравним время выполнения жадного алгоритма со временем точного алгоритма.

	$O(n!)$	$O(n^2)$
КОЛИЧЕСТВО СТАНЦИЙ	ТОЧНЫЙ АЛГОРИТМ	ЖАДНЫЙ АЛГОРИТМ
5	3.2 с	2.5 с
10	102.4 с	10 с
32	13.6 года	102.4 с
100	4×10^{21} года	16.67 мин

Упражнения

Для каждого из приведенных ниже алгоритмов укажите, является этот алгоритм жадным или нет.

8.3 Быстрая сортировка.

8.4 Поиск в ширину.

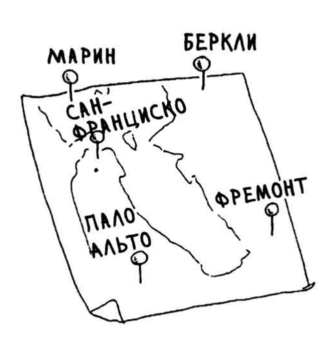
8.5 Алгоритм Дейкстры.

NP-полные задачи

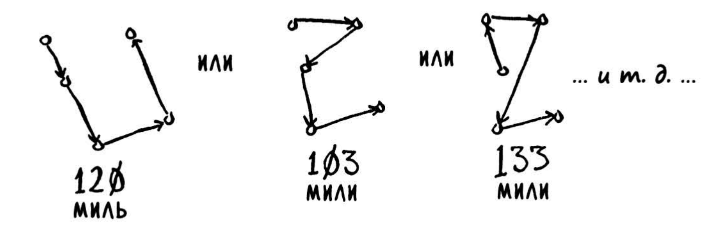
Для решения задачи о покрытии множества необходимо вычислить каждое возможное подмножество.



Вероятно, вы вспомнили задачу о коммивояжере из главы 1. В этой задаче коммивояжер должен был посетить пять разных городов.



Коммивояжер пытается найти кратчайший путь, который включит все пять городов. Чтобы найти кратчайший путь, сначала необходимо вычислить все возможные пути.



Сколько маршрутов необходимо вычислить для пяти городов?

Задача о коммивояжере — шаг за шагом

Начнем с малого. Допустим, городов всего два. Выбирать приходится всего из двух маршрутов.



Логично спросить: в задаче о коммивояжере существует ли конкретный город, с которого нужно начинать? Допустим, коммивояжер живет в Сан-Франциско и должен посетить еще четыре города. Сан-Франциско должен быть первым городом в маршруте.

Однако в каких-то ситуациях начальный город не задан. Допустим, вы работаете в курьерской службе FedEx и должны доставить пакет в пределах города. Пакет перевозится из Чикаго в один из 50 филиалов FedEx. Затем

пакет будет перегружен в машину, которая разъезжает по разным местам и доставляет пакеты. В какой филиал отгрузить пакет? На этот раз начальная точка неизвестна, и в задаче о коммивояжере вам придется вычислить как оптимальный путь, так и начальную точку.

Время выполнения обеих версий одинаково. Однако отсутствие определенного начального города упрощает пример, поэтому я выберу эту версию.

Два города = два возможных маршрута.

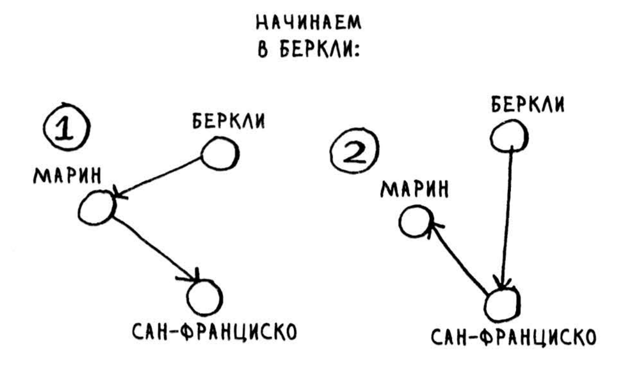
СКОЛЬКО МАРШРУТОВ?

На первый взгляд может показаться, что это один маршрут. Разве расстояние СФ>Марин не совпадает с расстоянием Марин>СФ? Не всегда. В некоторых городах (в том числе и в Сан-Франциско) много улиц с односторонним движением, и тогда вам не удается вернуться по тому пути, по которому вы приехали. Иногда приходится проехать лишнюю пару миль, чтобы найти выезд на шоссе. Так что эти два маршрута не всегда совпадают.

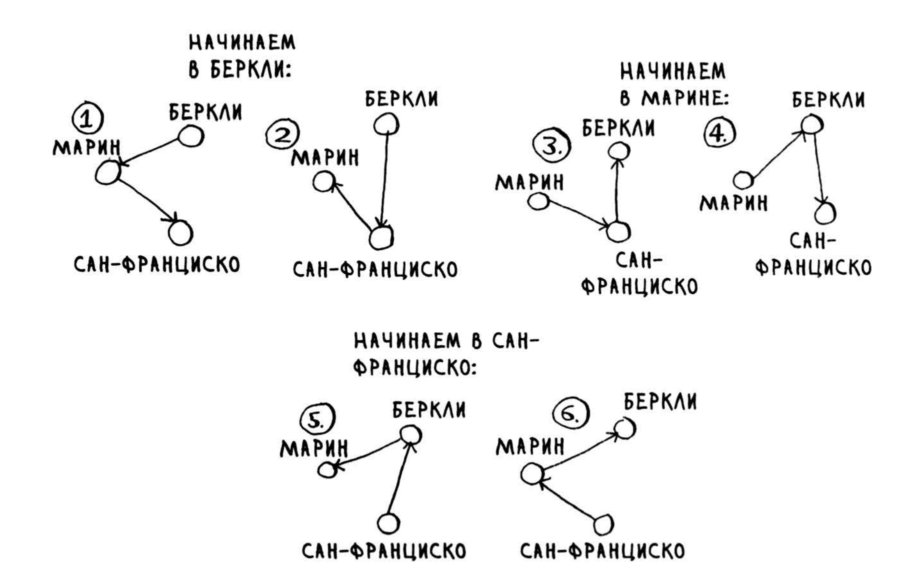
Три города

Теперь добавим к двум городам еще один. Сколько возможных маршрутов существует в этой конфигурации?

Если начать в Беркли, вы можете посетить два города.



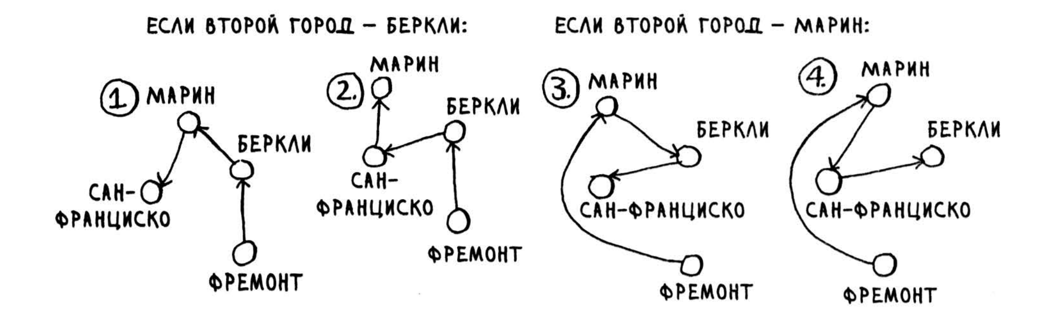
Всего шесть возможных маршрутов: по два для каждого города, с которого вы можете начать.

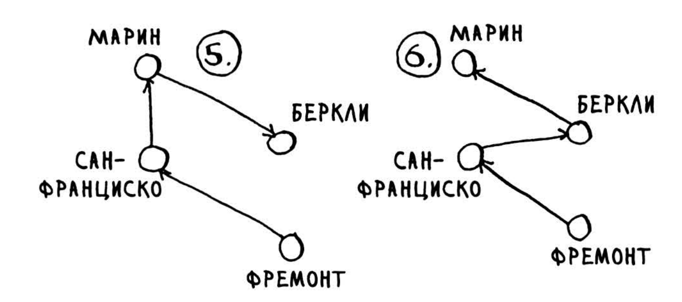


Итак, три города = шесть возможных маршрутов.

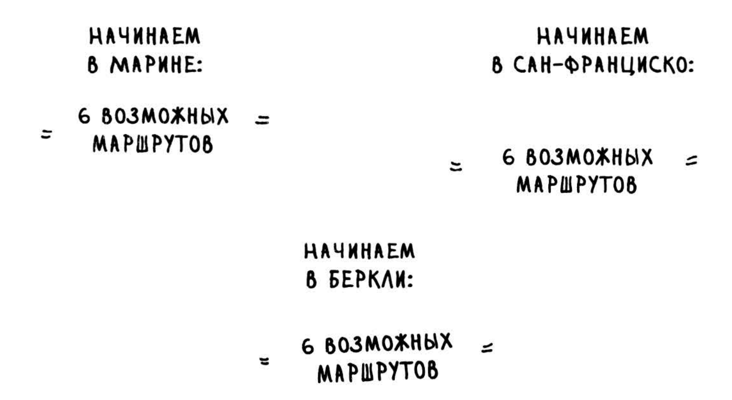
Четыре города

Добавим еще один город — Фремонт. Теперь допустим, что вы начали с Фремонта.





Мы знаем, что во Фремонте начинаются шесть возможных маршрутов. Ого! Да они очень похожи на шесть маршрутов, которые вы вычислили ранее, когда городов было всего три! Только теперь во всех маршрутах появился дополнительный город, Фремонт! Начинает проявляться закономерность. Предположим, из четырех городов выбирается начальный город Фремонт. Остается еще три города. И вы знаете, что для перемещения между тремя городами есть шесть разных маршрутов. Итак, если начать с Фремонта, существуют шесть возможных маршрутов. Также возможно начать с одного из других городов.



Четыре возможных начальных города, шесть возможных маршрутов для каждого начального города = 4 × 6 = 24 возможных маршрута.

Замечаете закономерность? Каждый раз, когда вы добавляете новый город, увеличивается количество вычисляемых маршрутов.

КОЛИЧЕСТВО ГОРОДОВ
1 → 1 МАРШРУТ
2 → 2 НАЧАЛЬНЫХ ГОРОДА * 1 МАРШРУТ ДЛЯ КАЖДОГО НАЧАЛА = 2 МАРШРУТА
3 → 3 НАЧАЛЬНЫХ ГОРОДА * 2 МАРШРУТА = 6 МАРШРУТОВ
4 → 4 НАЧАЛЬНЫХ ГОРОДА * 6 МАРШРУТОВ = 24 МАРШРУТА
5 → 5 НАЧАЛЬНЫХ ГОРОДОВ * 24 МАРШРУТА = 120 МАРШРУТОВ

Сколько возможных маршрутов существует для шести городов? 720, говорите? Да, вы правы. 5040 для 7 городов, 40 320 для 8 городов.

Такая зависимость называется *факториальной* (помните, что об этом говорилось в главе 3?) Итак, 5! = 120. Допустим, есть 10 городов. Сколько существует возможных маршрутов? 10! = 3 628 800. Уже для 10 городов приходится вычислять более 3 *миллионов* возможных маршрутов. Как видите, количество возможных маршрутов стремительно растет! Вот почему невозможно вычислить «правильное» решение задачи о коммивояжере при очень большом количестве городов.

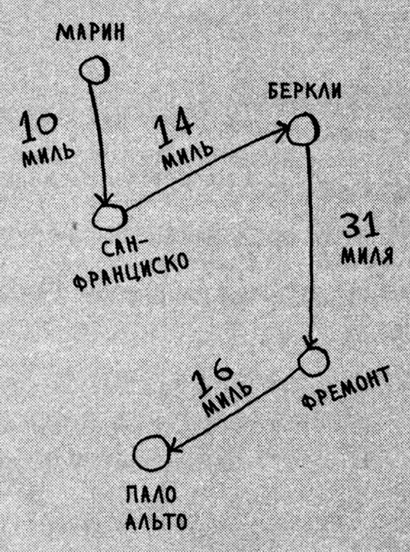
У задачи о коммивояжере и задаче покрытия множества есть кое-что общее: вы вычисляете каждое возможное решение и выбираете кратчайшее/минимальное. Обе эти задачи являются *NP-полными*.

ПРИБЛИЖЕННОЕ РЕШЕНИЕ

Как выглядит хороший приближенный алгоритм для задачи о коммивояжере? Это должен быть простой алгоритм, находящий короткий путь. Попробуйте самостоятельно найти ответ, прежде чем продолжить чтение.

Я бы сделал это примерно так: начальный город выбирается произвольно, после чего каждый раз, когда коммивояжер выбирает следу-

ющий город, он перемещается в ближайший город из тех, что он еще не посещал. Допустим, он начинает в Марине.



Суммарное расстояние — 71 миля. Может, это не самый короткий путь, но он достаточно близок к нему.

Короткое объяснение NP-полноты: некоторые задачи прославились сложностью своего решения. Задача о коммивояжере и задача о покрытии множества — два классических примера. Многие эксперты считают, что написать быстрый алгоритм для решения таких задач невозможно.

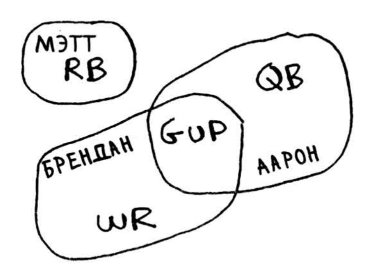
Как определить, что задача является NP-полной?



Джон подбирает игроков для своей команды по американскому футболу. У него есть список нужных качеств: хорошо играет в нападении, хорошо играет в защите, хорошо играет под дождем, хорошо играет под давлением и т. д. Также имеется список игроков, в котором каждый игрок обладает определенными качествами.

ИГРОК	КАЧЕСТВА
МЭТТ ФОРТЕ	RB
БРЕНДАН МАРШАЛЛ	WR / ХОРОШО ИГРАЕТ ПОД ДАВЛЕНИЕМ
ААРОН РОДЖЕРС	QB / ХОРОШО ИГРАЕТ ПОД ДАВЛЕНИЕМ
...	...

Джон хочет подобрать команду, которая обладает полным набором качеств, но размер команды ограничен. «Минутку, — осознает Джон, — но ведь это задача покрытия множества!»



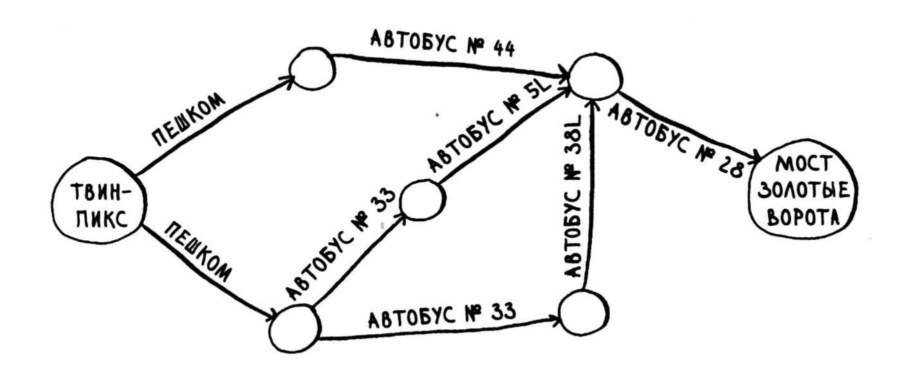
Для создания команды Джон может воспользоваться тем же приближенным алгоритмом:

1. Найти игрока с большинством качеств, которые еще не были реализованы.

2. Повторять до тех пор, пока не будут реализованы все качества (или пока не кончатся свободные места в команде).

NP-полные задачи встречаются очень часто. И было бы полезно, если бы вы могли понять, что решаемая задача является NP-полной. В этот момент можно прекратить поиски идеального решения и перейти к решению с применением приближенного алгоритма. Но определить, является ли ваша задача NP-полной, непросто. Обычно различия между легко решаемыми и NP-полными задачами весьма незначительны. Например, в предыдущих

главах я много говорил о кратчайших путях. Вы знаете, как вычислить кратчайший путь из точки A в точку B.



Но если вы хотите найти кратчайший путь, соединяющий несколько точек, то это уже задача о коммивояжере, которая является NP-полной. Короче говоря, не существует простого способа определить, является ли задача, с которой вы работаете, NP-полной. Несколько характерных признаков:

- ваш алгоритм быстро работает при малом количестве элементов, но сильно замедляется при увеличении их числа;
- формулировка «все комбинации X» часто указывает на NP-полноту задачи;
- вам приходится вычислять все возможные варианты X, потому что задачу невозможно разбить на меньшие подзадачи? Такая задача может оказаться NP-полной;
- если в задаче встречается некоторая последовательность (например, последовательность городов, как в задаче о коммивояжере) и задача не имеет простого решения, она может оказаться NP-полной;
- если в задаче встречается некоторое множество (например, множество радиостанций) и задача не имеет простого решения, она может оказаться NP-полной;
- можно ли переформулировать задачу в условиях задачи покрытия множества или задачи о коммивояжере? В таком случае ваша задача определенно является NP-полной.

Упражнения

8.6 Почтальон должен доставить письма в 20 домов. Ему нужно найти кратчайший путь, проходящий через все 20 домов. Является ли эта задача NP-полной?

8.7 Имеется задача поиска максимальной *клики* в множестве людей (кликой называется множество людей, каждый из которых знаком со всеми остальными). Является ли эта задача NP-полной?

8.8 Вы рисуете карту США, на которой два соседних штата не могут быть окрашены в одинаковый цвет. Требуется найти минимальное количество цветов, при котором любые два соседних штата будут окрашены в разные цвета. Является ли эта задача NP-полной?

Шпаргалка

- Жадные алгоритмы стремятся к локальной оптимизации в расчете на то, что в итоге будет достигнут глобальный оптимум.
- У NP-полных задач не существует известных быстрых решений.
- Если у вас имеется NP-полная задача, лучше всего воспользоваться приближенным алгоритмом.
- Жадные алгоритмы легко реализуются и быстро выполняются, поэтому из них получаются хорошие приближенные алгоритмы.



Динамическое программирование



В этой главе

- ✓ Вы освоите динамическое программирование — метод решения сложных задач, разбиваемых на подзадачи, которые решаются в первую очередь.
- ✓ Рассматриваются примеры, которые научат вас искать решения новых задач, основанные на методе динамического программирования.

Задача о рюкзаке

Вернемся к задаче о рюкзаке из главы 8. У вас есть рюкзак, в котором можно унести товары общим весом до 4 фунтов.





Есть три предмета, которые можно уложить в рюкзак.

Какие предметы следует положить в рюкзак, чтобы стоимость добычи была максимальной?

Простое решение

Простой алгоритм выглядит так: вы перебираете все возможные множества товаров и находите множество с максимальной стоимостью.

ПУСТО
$0

ГИТАРА
$1500

МАГНИТОФОН
$3000

НОУТБУК
$2000

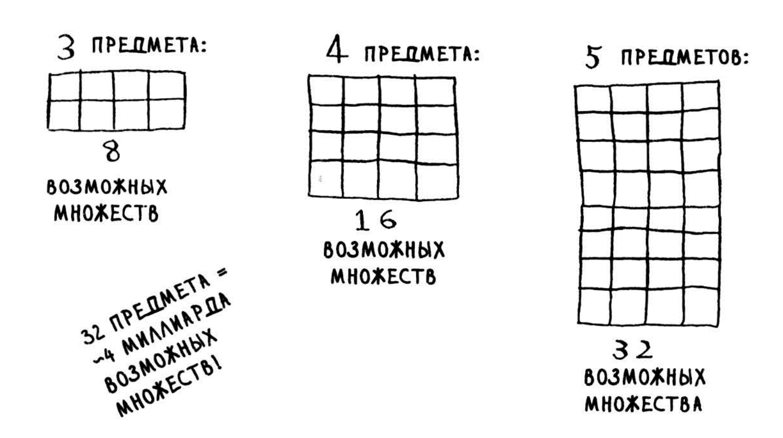
ГИТАРА + МАГНИТОФОН
НЕ ПОМЕЩАЕТСЯ

ГИТАРА + НОУТБУК
$3500
МАКСИМАЛЬНАЯ СТОИМОСТЬ

МАГНИТОФОН + НОУТБУК
НЕ ПОМЕЩАЕТСЯ

ГИТАРА + МАГНИТОФОН + НОУТБУК
НЕ ПОМЕЩАЕТСЯ

Такое решение работает, но очень медленно. Для 3 предметов приходится обработать 8 возможных множеств, для 4 — 16 и т. д. С каждым добавляемым предметом количество множеств удваивается! Этот алгоритм выполняется за время $O(2^n)$, что очень, очень медленно.



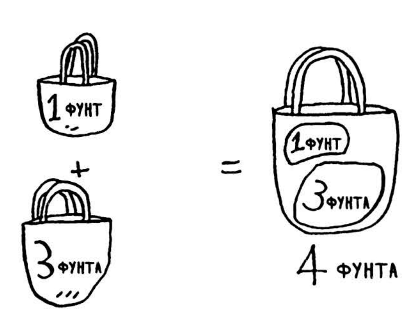
Для любого сколько-нибудь значительного количества предметов это неприемлемо. В главе 8 вы видели, как вычисляются *приближенные* решения. Такие решения близки к оптимальным, но могут не совпадать с ними.

Как же вычислить оптимальное решение?

Динамическое программирование

Ответ: с помощью динамического программирования! Давайте посмотрим, как работает этот метод. Процедура начинается с решения подзадач с постепенным переходом к решению полной задачи.

В задаче о рюкзаке начать следует с решения задачи для меньшего рюкзака (или «подрюкзака»), а потом на этой основе попытаться решить исходную задачу.



Динамическое программирование — достаточно сложная концепция; не огорчайтесь, если после первого прочтения что-то останется непонятным. Примеры помогут вам разобраться в теме.

Для начала я покажу вам алгоритм в действии. После этого у вас наверняка появится много вопросов! Я постараюсь ответить на них.

Каждый алгоритм динамического программирования начинается с таблицы. Вот как выглядит таблица для задачи о рюкзаке.



Строки таблицы представляют предметы, а столбцы — емкость рюкзака от 1 до 4 фунтов. Все эти столбцы нужны, потому что они упрощают вычисление стоимостей «подрюкзаков».

В исходном состоянии таблица пуста. Нам предстоит заполнить каждую ячейку таблицы. После того как таблица будет заполнена, вы получите ответ на свою задачу. Пожалуйста, внимательно разберитесь в происходящем. Нарисуйте собственную таблицу, а мы вместе ее заполним.

Строка Гитара

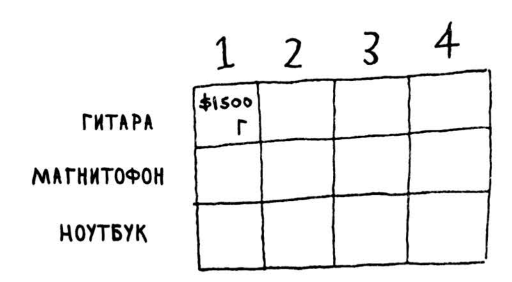
Точная формула для вычисления значений в таблице будет приведена позднее, а пока ограничимся общим описанием. Начнем с первой строки.

	1	2	3	4
ГИТАРА				
МАГНИТОФОН				
НОУТБУК				

Строка снабжена пометкой «гитара»; это означает, что вы пытаетесь уложить гитару в рюкзак. В каждой ячейке принимается простое решение: класть гитару в рюкзак или нет? Помните: мы пытаемся найти множество элементов с максимальной стоимостью.

В первой ячейке емкость рюкзака равна 1 фунту. Гитара также весит 1 фунт — значит, она поместится в рюкзак! Итак, стоимость этой ячейки составляет $1500, а в рюкзаке лежит гитара.

Начнем заполнять ячейку.



По тому же принципу каждая ячейка в таблице содержит список всех элементов, которые помещаются в рюкзаке на данный момент.

Посмотрим на следующую ячейку. На этот раз емкость рюкзака составляет 2 фунта. Понятно, что гитара здесь поместится!

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г		
МАГНИТОФОН				
НОУТБУК				

Процедура повторяется для остальных ячеек строки. Вспомните, что текущей является первая строка, поэтому выбирать приходится *только* из одного предмета — гитары. Считайте, что два других предмета пока недоступны.

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН				
НОУТБУК				

Возможно, к этому моменту вы слегка сбиты с толку. Почему все это делается для рюкзаков с емкостью 1, 2 и т. д., если в задаче речь идет о рюкзаке с емкостью 4 фунта? Помните, что я говорил ранее? Метод динамического программирования начинает с малых задач, а затем переходит к большой задаче. Вы решаете подзадачи, которые помогут в решении большой задачи. Читайте дальше, и ситуация постепенно прояснится.

После того как первая строка будет заполнена, таблица будет выглядеть так:

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН				
НОУТБУК				

Помните, что мы стремимся обеспечить максимальную стоимость предметов в рюкзаке. *Эта строка представляет текущую лучшую оценку максимума.* Итак, на данный момент из этой строки следует, что для рюкзака с емкостью 4 фунта максимальная стоимость предметов составит $1500.

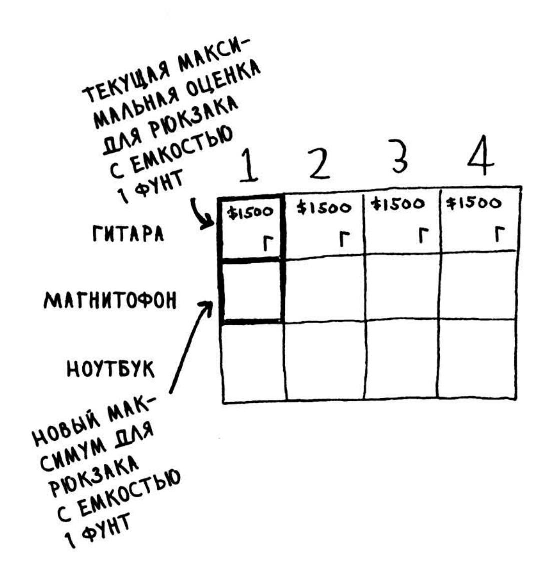
	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН				
НОУТБУК				

НАША ТЕКУЩАЯ ОЦЕНКА ТОГО, ЧТО СЛЕДУЕТ КРАСТЬ: ГИТАРУ ЗА $1500

Вы знаете, что это решение неокончательно. В процессе работы алгоритма оценка будет уточняться.

Магнитофон

Займемся следующей строкой, которая относится к магнитофону. Теперь, когда вы перешли ко второй строке, появляется выбор между магнитофоном и гитарой. В каждой строке можно взять предмет этой строки или предметы, находящиеся в верхних строках. Таким образом, сейчас нельзя выбрать ноутбук, но можно выбрать магнитофон и/или гитару. Начнем с первой ячейки (рюкзак с емкостью 1 фунт). Текущая максимальная стоимость предметов, которые можно положить в рюкзак с емкостью 1 фунт, составляет $1500.



Брать магнитофон или нет?

Емкость рюкзака составляет 1 фунт. Поместится туда магнитофон? Нет, он слишком тяжел! Так как магнитофон не помещается в рюкзак, максимальная оценка для 1-фунтового рюкзака *остается* равной $1500.

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	↓ $1500 Г			
НОУТБУК				

То же самое происходит со следующими двумя клетками. Емкость этих рюкзаков составляет 2 и 3 фунта соответственно. Старая максимальная стоимость для обеих ячеек была равна $1500.

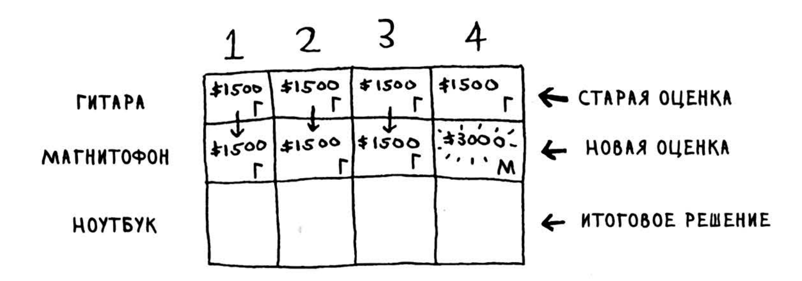
	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	↓ $1500 Г	↓ $1500 Г	↓ $1500 Г	
НОУТБУК				

Магнитофон все равно не помещается, так что оценка остается неизменной.

А если емкость рюкзака увеличивается до 4 фунтов? Ага, магнитофон наконец-то войдет в рюкзак! Старая максимальная стоимость была равна $1500, но если вместо гитары положить магнитофон, она увеличится до $3000! Берем магнитофон.

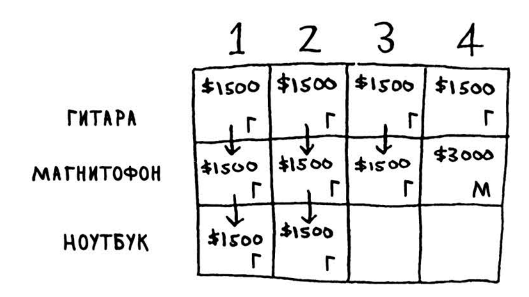
	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	↓ $1500 Г	↓ $1500 Г	↓ $1500 Г	$3000 М
НОУТБУК				

Оценка только что обновилась! Имея рюкзак емкостью 4 фунта, вы можете положить в него товары стоимостью по крайней мере $3000. Из таблицы видно, что оценка постепенно возрастает.

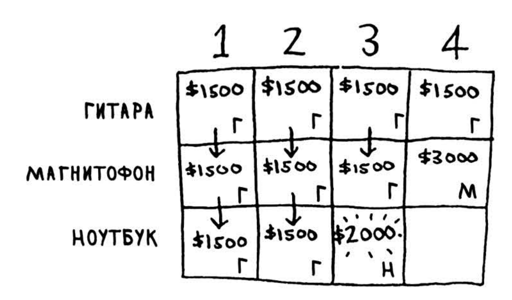


Ноутбук

А теперь проделаем то же для ноутбука! Ноутбук весит 3 фунта, поэтому он не поместится в рюкзак с емкостью 1 или 2 фунта. Оценка для первых двух ячеек остается на уровне $1500.



Для 3 фунтов старая оценка составляла $1500. Но теперь вы можете выбрать ноутбук, который стоит $2000. Следовательно, новая максимальная оценка равна $2000!



При 4 фунтах ситуация становится по-настоящему интересной. Это очень важная часть. В настоящее время оценка составляет $3000. В рюкзак можно положить ноутбук, но он стоит всего $2000.

$3000 МАГНИТОФОН или $2000 НОУТБУК

Так-так, старая оценка была лучше. Но постойте! Ноутбук весит всего 3 фунта, так что 1 фунт еще свободен! На это место можно еще что-нибудь положить.

$3000 МАГНИТОФОН или ($2000 НОУТБУК + ??? СВОБОДНОЕ МЕСТО НА 1 ФУНТ)

Какую максимальную стоимость можно разместить в 1 фунте? Да вы же уже вычислили ее!

1	2	3	4
$1500 Г	$1500 Г	$1500 Г	$1500 Г
$1500 Г	$1500 Г	$1500 Г	$3000 М
$1500 Г	$1500 Г	$2000 Н	

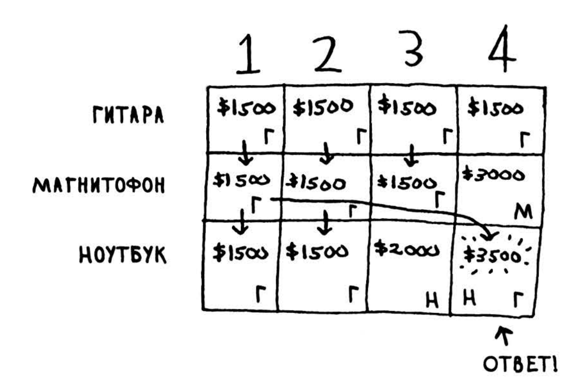
МАКСИМАЛЬНАЯ СТОИМОСТЬ ДЛЯ 1 ФУНТА →

В соответствии с последней оценкой в свободном месте емкостью в 1 фунт можно разместить гитару стоимостью $1500. Следовательно, настоящее сравнение выглядит так:

$3000 МАГНИТОФОН или ($2000 НОУТБУК + $1500 ГИТАРА)

Вы удивлялись, зачем мы вычисляем максимальную стоимость для рюкзаков меньшей емкости? Надеюсь, теперь все стало на свои места! Если в рюкзаке остается свободное место, вы можете использовать ответы на эти подзадачи для определения того, чем заполнить это пространство. Вместо магнитофона лучше взять ноутбук + гитару за $3500.

В завершающем состоянии таблица выглядит так:



Итак, мы получили ответ: максимальная стоимость товаров, которые поместятся в рюкзак, равна $3500 — для гитары и ноутбука.

Возможно, вы подумали, что я воспользовался другой формулой для вычисления стоимости последней ячейки. Это связано с тем, что я опустил некоторые лишние сложности при заполнении предыдущих ячеек. Стоимость каждой ячейки вычисляется по постоянной формуле, которая выглядит так:

СТРОКА СТОЛБЕЦ

CELL[i][j] = МАКСИМУМ {
1. ПРЕДЫДУЩИЙ МАКСИМУМ (ЗНАЧЕНИЕ В CELL[i-1][j])
ИЛИ
2. СТОИМОСТЬ ТЕКУЩЕГО ЭЛЕМЕНТА + СТОИМОСТЬ ОСТАВШЕГОСЯ ПРОСТРАНСТВА
CELL[i-1][j – ВЕС ПРЕДМЕТА]

Применяя эту формулу к каждой ячейке таблицы, вы получите такую же таблицу, как у меня. Помните, что я говорил о решении подзадач?

Вы объединили решения двух подзадач для решения еще одной, большей задачи.



Задача о рюкзаке: вопросы



Вам все еще кажется, что это какой-то фокус? В этом разделе я отвечу на некоторые часто задаваемые вопросы.

Что произойдет при добавлении элемента?

Представьте, что вы увидели четвертый предмет, который тоже можно засунуть в рюкзак! Вместе со всем предыдущим добром можно также украсть iPhone.

Придется ли пересчитывать все заново с новым предметом? Нет. Напомню, что динамическое программирование последовательно строит решение на основании вашей оценки. К настоящему моменту максимальные стоимости выглядят так:

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	$1500 Г	$1500 Г	$1500 Г	$3000 М
НОУТБУК	$1500 Г	$1500 Г	$2000 Н	$3500 Н Г

Это означает, что в рюкзак с емкостью 4 фунта можно упаковать товары стоимостью до $3500. И вы полагали, что это итоговый максимум. Но давайте добавим новую строку для iPhone.

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	$1500 Г	$1500 Г	$1500 Г	$3000 М
НОУТБУК	$1500 Г	$1500 Г	$2000 Н	$3500 Н Г
IPHONE				

НОВЫЙ ОТВЕТ

Оказывается, в таблице появляется новый максимум! Попробуйте заполнить последнюю строку, прежде чем читать дальше.

Начнем с первой ячейки. iPhone сам по себе помещается в рюкзак с емкостью 1 фунт. Старый максимум был равен $1500, но iPhone стоит $2000. Значит, берем iPhone.

	1	2	3	4
ГИТАРА	$1500 Г	$1500 Г	$1500 Г	$1500 Г
МАГНИТОФОН	$1500 Г	$1500 Г	$1500 Г	$3000 М
НОУТБУК	$1500 Г	$1500 Г	$2000 Н	$3500 Н Г
IPHONE	$2000 I			

В следующей ячейке можно разместить iPhone *и* гитару.

$1500 Г	$1500 Г	$1500 Г	$1500 Г
$1500 Г	$1500 Г	$1500 Г	$3000 М
$1500 Г	$1500 Г	$2000 Н	$3500 Н Г
$2000 I	$3500 I Г		

Для ячейки 3 ничего лучшего, чем снова взять iPhone вместе с гитарой, все равно не найдется, поэтому оставим этот вариант.

А вот в последней ячейке ситуация становится более интересной. Текущий максимум равен $3500. Вы снова можете взять iPhone, и у вас еще останется свободное место на 3 фунта.

$3500 НОУТБУК+ГИТАРА ИЛИ ($2000 IPHONE + ??? СВОБОДНОЕ МЕСТО НА 3 ФУНТА)

Но эти 3 фунта можно заполнить на $2000! $2000 от iPhone + $2000 из старой подзадачи: получается $4000. Новый максимум!

Вот как выглядит новая завершающая таблица.

$1500 Г	$1500 Г	$1500 Г	$1500 Г
$1500 Г	$1500 Г	$1500 Г	$3000 М
$1500 Г	$1500 Г	$2000 Н	$3500 Н Г
$3500 I	$3500 I Г	$3500 I Г	$4000 I Н

НОВЫЙ ОТВЕТ

Вопрос: может ли значение в столбце *уменьшиться*? Такое возможно?

МАКСИМАЛЬНАЯ СТОИМОСТЬ УМЕНЬШАЕТСЯ В ПРОЦЕССЕ РАБОТЫ

1	2	3	4
$1500	$1500	$1500	$1500
∅	∅	∅	$3000

Подумайте над ответом, прежде чем продолжить чтение.

Ответ: нет. При каждой итерации сохраняется текущая оценка максимума. Эта оценка ни при каких условиях не может быть меньше предыдущей!

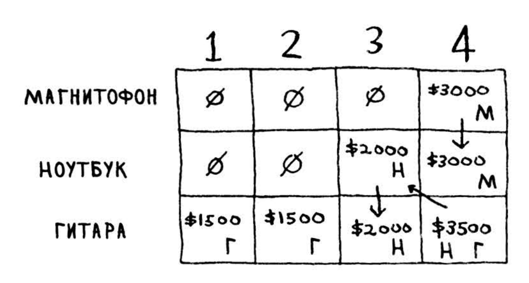
Упражнения

9.1 Предположим, к предметам добавился еще один: MP3-плеер. Он весит 1 фунт и стоит $1000. Стоит ли брать его?

Что произойдет при изменении порядка строк?

Изменится ли ответ? Допустим, строки заполняются в другом порядке: магнитофон, ноутбук, гитара. Как будет выглядеть таблица? Заполните таблицу самостоятельно, прежде чем двигаться дальше.

Таблица должна выглядеть так:



	1	2	3	4
МАГНИТОФОН	∅	∅	∅	$3000 М
НОУТБУК	∅	∅	$2000 Н	$3000 М
ГИТАРА	$1500 Г	$1500 Г	$2000 Н	$3500 Н Г

Ответ не изменился. Он не зависит от порядка строк.

Можно ли заполнять таблицу по столбцам, а не по строкам?

Попробуйте сами! В данной задаче это ни на что не влияет, но в других задачах возможны изменения.

Что произойдет при добавлении меньшего элемента?

Допустим, вы можете выбрать ожерелье, которое весит 0,5 фунта и стоит $1000. Пока структура таблицы предполагает, что все веса являются целыми числами. Теперь вы решаете взять ожерелье. Остается еще 3,5 фунта. Какую максимальную стоимость можно разместить в объеме 3,5 фунта? Неизвестно! Вы вычисляли стоимость только для рюкзаков с емкостью 1, 2, 3 и 4 фунта. Теперь придется определять стоимость для рюкзака на 3,5 фунта.

Из-за ожерелья приходится повысить точность представления весов, поэтому таблица должна измениться.

	0.5	1	1.5	2	2.5	3	3.5	4
ГИТАРА								
МАГНИТОФОН								
НОУТБУК								
ОЖЕРЕЛЬЕ								

Можно ли взять часть предмета?

Допустим, вы наполняете рюкзак в продуктовом магазине. Вы можете украсть мешки с чечевицей и рисом. Если весь мешок не помещается, его можно открыть и отсыпать столько, сколько унесете. В этом случае вы уже не действуете по принципу «все или ничего» — можно взять только часть

предмета. Как решить такую задачу методом динамического программирования?

Ответ: никак. В решении, полученном методом динамического программирования, вы либо берете предмет, либо не берете. Алгоритм не предусматривает возможность взять половину предмета.

Однако проблема легко решается с помощью жадного алгоритма! Сначала вы берете самый ценный предмет — настолько бóльшую его часть, насколько возможно. Когда самый ценный предмет будет исчерпан, вы берете максимально возможную часть следующего по ценности предмета и т. д.

Допустим, вы можете выбирать из следующих товаров.



Фунт киноа стоит дороже, чем фунт любого другого товара. А раз так — набирайте столько киноа, сколько сможете унести! И если вам удастся набить им свой рюкзак, то это и будет лучшее из возможных решений.



Если киноа кончится, а в рюкзаке еще остается свободное место, возьмите следующий по ценности товар и т. д.

Оптимизация туристического маршрута

Представьте, что вы приехали в Лондон на выходные. У вас два дня, а мест, которые хочется посетить, слишком много. Побывать везде не получится, поэтому вы составляете список.

ДОСТОПРИМЕЧАТЕЛЬНОСТЬ	ВРЕМЯ	ОЦЕНКА
ВЕСТМИНСТЕРСКОЕ АББАТСТВО	½ ДНЯ	7
ТЕАТР «ГЛОБУС»	½ ДНЯ	6
НАЦИОНАЛЬНАЯ ГАЛЕРЕЯ	1 ДЕНЬ	9
БРИТАНСКИЙ МУЗЕЙ	2 ДНЯ	9
СОБОР СВ. ПАВЛА	½ ДНЯ	8

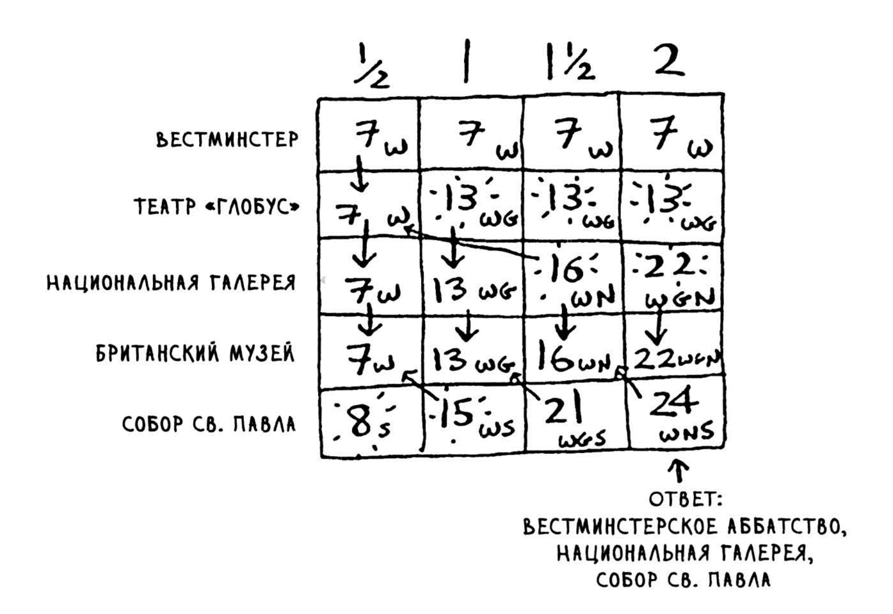
Для каждой достопримечательности, которую вы захотите увидеть, вы указываете, сколько времени займет осмотр и насколько сильно вы хотите ее увидеть. Сможете ли вы построить оптимальный туристический маршрут на основании этого списка?

Да это все та же задача о рюкзаке! Вместо ограниченной емкости рюкзака — ограниченное время. Вместо магнитофонов и ноутбуков — список мест, которые вы хотите посетить. Нарисуйте таблицу динамического программирования для списка, прежде чем двигаться дальше.

Вот как должна выглядеть эта таблица:

	½	1	1½	2
ВЕСТМИНСТЕР				
ТЕАТР «ГЛОБУС»				
НАЦИОНАЛЬНАЯ ГАЛЕРЕЯ				
БРИТАНСКИЙ МУЗЕЙ				
СОБОР СВ. ПАВЛА				

Вы изобразили ее правильно? Теперь заполните. Какие достопримечательности вы выберете? Ответ:



Взаимозависимые элементы

Предположим, вы хотите посетить Париж и добавили в свой список пару элементов.

ЭЙФЕЛЕВА БАШНЯ	1½ DAY	8
ЛУВР	1½ DAY	9
НОТР-ДАМ	1½ DAY	7

На их посещение потребуется много времени, потому что сначала придется приехать из Лондона в Париж. Переезд отнимает полдня. Если вы захотите посмотреть все 3 достопримечательности, осмотр займет 4,5 дня.

Стоп, небольшая поправка. Вам не обязательно приезжать в Париж ради каждой достопримечательности. После того как вы там окажетесь, каждый

последующий элемент займет всего один день. Следовательно, потребуется 1 день на каждую достопримечательность + 1 день на переезды = 3,5 дня, а не 4,5.

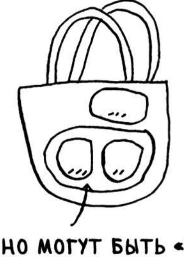
Если вы положите Эйфелеву башню в свой «рюкзак», то Лувр станет «дешевле» — он займет всего 1 день вместо 1,5 дня. Как смоделировать это обстоятельство в динамическом программировании?

Никак. Динамическое программирование — мощный метод, способный решать подзадачи и использовать полученные ответы для решения большой задачи. *Динамическое программирование работает только в том случае, если каждая подзадача автономна, то есть не зависит от других подзадач.* Из этого следует, что учесть поездки в Париж в алгоритме динамического программирования не удастся.

Может ли оказаться, что решение требует более двух «подрюкзаков»?

Может оказаться, что в лучшем решении должны отбираться больше двух элементов. В текущем варианте алгоритма объединяются не более двух «подрюкзаков» — больше двух их не бывает. Однако вполне возможно, что у этих «подрюкзаков» будут собственные «подрюкзаки».





Возможно ли, что при лучшем решении в рюкзаке остается пустое место?



Да. Представьте, что вы можете также положить в рюкзак бриллиант.

Бриллиант очень крупный: он весит 3,5 фунта и стоит 1 миллион долларов — намного больше, чем любые другие предметы. Безусловно, нужно брать именно его! Но в рюкзаке остается еще пустое место на 0,5 фунта, и в нем ничего не поместится.

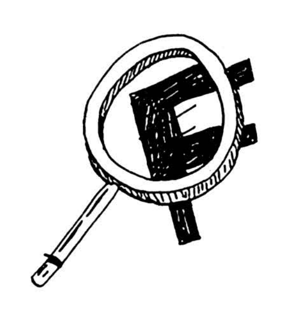
Упражнения

9.2 Предположим, что вы собираетесь в турпоход. Емкость вашего рюкзака составляет 6 фунтов, и вы можете взять предметы из следующего списка. У каждого предмета имеется стоимость; чем она выше, тем важнее предмет:

- вода, 3 фунта, 10;
- книга, 1 фунт, 3;
- еда, 2 фунта, 9;
- куртка, 2 фунта, 5;
- камера, 1 фунт, 6

Как выглядит оптимальный набор предметов для похода?

Самая длинная общая подстрока



Мы рассмотрели одну задачу динамического программирования. Какие выводы из нее можно сделать?

- Динамическое программирование применяется для оптимизации какой-либо характеристики

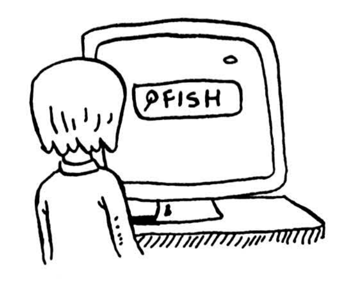
при заданных ограничениях. В задаче о рюкзаке требуется максимизировать стоимость отобранных предметов с ограничениями по емкости рюкзака.

- Динамическое программирование работает только в ситуациях, в которых задача может быть разбита на автономные подзадачи, не зависящие друг от друга.

Построить решение на базе динамического программирования бывает непросто. В этом разделе мы сосредоточимся на этой теме. Несколько общих рекомендаций:

- в каждом решении из области динамического программирования строится таблица;
- значения ячеек таблицы обычно соответствуют оптимизируемой характеристике. Для задачи о рюкзаке значения представляли общую стоимость товаров;
- каждая ячейка представляет подзадачу, поэтому вы должны подумать о том, как разбить задачу на подзадачи. Это поможет вам определиться с осями.

Рассмотрим еще один пример. Допустим, вы открыли сайт *dictionary.com*. Пользователь вводит слово, а сайт возвращает определение. Но если пользователь ввел несуществующее слово, нужно предположить, какое слово имелось в виду. Алекс ищет определение «fish», но он случайно ввел «hish». Такого слова в словаре нет, но зато у вас есть список похожих слов.



СЛОВА, ПОХОЖИЕ НА "HISH":

- FISH
- VISTA

(Это несерьезный пример, поэтому список ограничен всего двумя словами. Вероятно, на практике такой список будет состоять из тысяч слов.)

Итак, Алекс ввел строку *hish*. Какое слово он хотел ввести на самом деле: *fish* или *vista*?

Построение таблицы

Как должна выглядеть таблица для этой задачи? Вы должны ответить на следующие вопросы.

- Какие значения должны содержаться в ячейках?
- Как разбить эту задачу на подзадачи?
- Каков смысл осей таблицы?

В динамическом программировании вы пытаетесь максимизировать некоторую характеристику. В данном случае ищется самая длинная подстрока, общая в двух словах. Какую общую подстроку содержат *hish* и *fish*? А как насчет *hish* и *vista*? Именно это требуется вычислить.

Как говорилось ранее, значения в ячейках обычно представляют ту характеристику, которую вы пытаетесь оптимизировать. Вероятно, в данном случае этой характеристикой будет число: длина самой длинной подстроки, общей для двух строк.

Как разделить эту задачу на подзадачи? Например, можно заняться сравнением подстрок. Вместо того чтобы сравнивать *hish* и *fish*, можно сначала сравнить *his* и *fis*. Каждая ячейка будет содержать длину самой длинной подстроки, общей для двух подстрок. Такое решение также подсказывает, что строками и столбцами таблицы, вероятно, будут два слова. А значит, таблица будет выглядеть примерно так:

	H	I	S	H
F				
I				
S				
H				

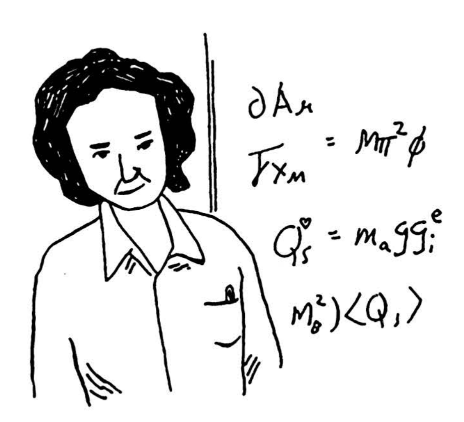
Если у вас голова идет кругом, не огорчайтесь. Это сложный материал — собственно, именно поэтому я объясняю его в конце книги! Ниже будет приведено упражнение, чтобы вы могли самостоятельно потренироваться в динамическом программировании.

Заполнение таблицы

Сейчас вы уже достаточно хорошо представляете, как должна выглядеть таблица. По какой формуле заполняются ячейки таблицы? Мы можем немного упростить свою задачу, потому что уже знаем решение — у *hish* и *fish* имеется общая подстрока длины 3: *ish*.

Однако этот факт ничего не говорит о том, какая формула должна использоваться. Программисты иногда шутят об использовании алгоритма Фейнмана. *Алгоритм Фейнмана*, названный по имени известного физика Ричарда Фейнмана, работает так:

1. Записать формулировку задачи.
2. Хорошенько подумать.
3. Записать решение.

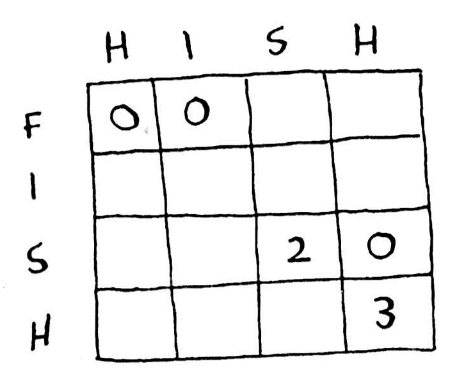


Да, программисты — большие шутники!

По правде говоря, простого способа вычислить формулу для данного случая не существует. Вам придется экспериментировать и искать работоспособное

решение. Иногда алгоритм предоставляет не точный рецепт, а основу, на которую вы наращиваете свою идею.

Попробуйте предложить решение этой задачи самостоятельно. Даю подсказку — часть таблицы выглядит так:



Чему равны другие значения? Вспомните, что каждая ячейка содержит значение *подзадачи*. Почему ячейка (3, 3) содержит значение 2? Почему ячейка (3, 4) содержит значение 0?

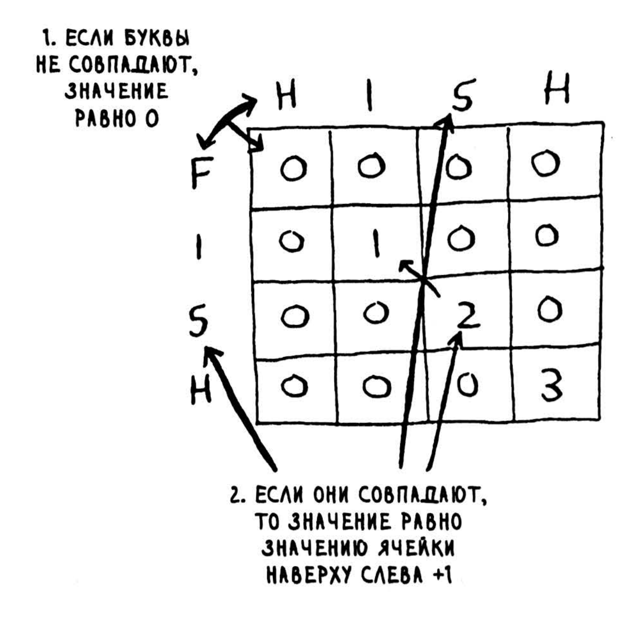
Попытайтесь вывести формулу самостоятельно, прежде чем продолжить читать. Даже если вам не удастся получить правильный ответ, мои объяснения покажутся вам намного более понятными.

Решение

Итоговая версия таблицы выглядит так:

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

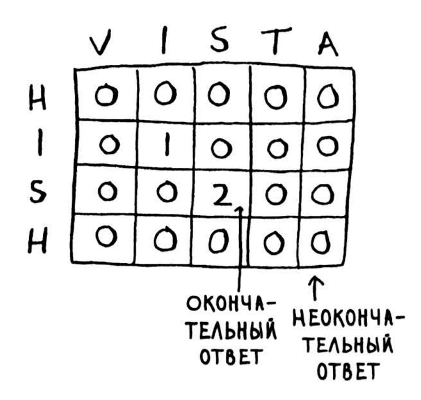
А это моя формула для заполнения ячеек:



На псевдокоде эта формула реализуется так:

```
if word_a[i] == word_b[j]:        ◄············ Буквы совпадают
  cell[i][j] = cell[i-1][j-1] + 1  ◄·········· Буквы не совпадают
else:
  cell[i][j] = 0
```

Аналогичная таблица для строк *hish* и *vista*:



Важный момент: в этой задаче окончательное решение далеко не всегда находится в последней ячейке! В задаче о рюкзаке последняя ячейка всегда содержит окончательное решение. Но в задаче поиска самой длинной общей подстроки решение определяется самым большим числом в таблице — и это может быть не последняя, а какая-то другая ячейка.

Вернемся к исходному вопросу: какая строка ближе к *hish*? У строк *hish* и *fish* есть общая подстрока длиной в три буквы. У *hish* и *vista* есть общая подстрока из двух букв. Скорее всего, Алекс хотел ввести строку *fish*.

Самая длинная общая подпоследовательность

Предположим, Алекс ввел строку *fosh*. Какое слово он имел в виду: *fish* или *fort*?

Сравним строки по формуле самой длинной общей подстроки.

	F	O	S	H
F	1	0	0	0
O	0	2	0	0
R	0	0	0	0
T	0	0	0	0

ИЛИ

	F	O	S	H
F	1	0	0	0
I	0	0	0	0
S	0	0	1	0
H	0	0	0	2

Длина подстрок одинакова: две буквы! Но *fosh* при этом ближе к *fish*:

FOSH → FISH = 3

FOSH → FORT = 2

Мы сравниваем самую длинную общую *подстроку*, а на самом деле нужно сравнивать самую длинную общую *подпоследовательность*: количество букв в последовательности, общих для двух слов. Как вычислить самую длинную общую подпоследовательность?

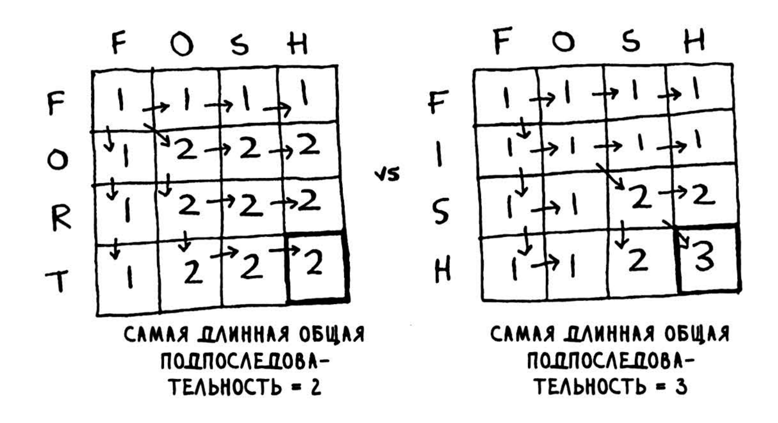
Ниже приведена частично заполненная таблица для *fish* и *fosh*.

	F	O	S	H
F	1	1		
I	1			
S		1	2	2
H				

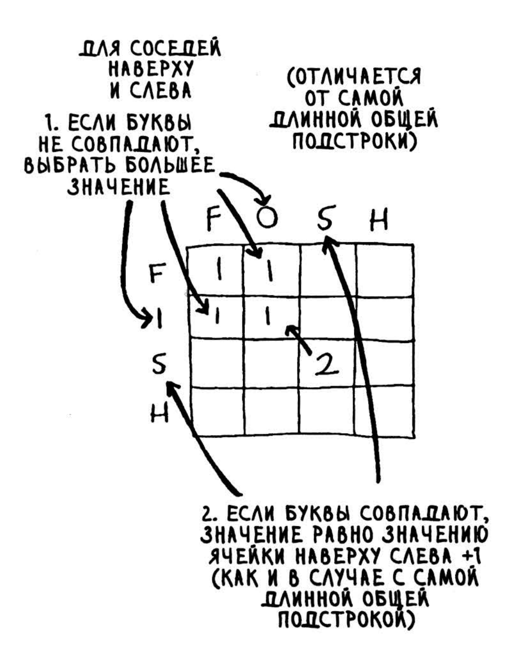
Сможете ли вы определить формулу для этой таблицы? Самая длинная общая подпоследовательность имеет много общего с самой длинной общей подстрокой, и их формулы тоже очень похожи. Попробуйте решить задачу самостоятельно, а я приведу ответ ниже.

Самая длинная общая подпоследовательность — решение

Окончательная версия таблицы:



А теперь моя формула для заполнения каждой ячейки:



На псевдокоде эта формула реализуется так:

```
if word_a[i] == word_b[j]:   <-------- Буквы совпадают
  cell[i][j] = cell[i-1][j-1] + 1
else:  <-------------------------------- Буквы не совпадают
  cell[i][j] = m a x(cell[i-1][j], cell[i][j-1])
```

Поздравляю — вы справились! Безусловно, это была одна из самых сложных глав в книге. Находит ли динамическое программирование практическое применение? Да, находит.

- Биологи используют самую длинную общую подпоследовательность для выявления сходства в цепях ДНК. По этой метрике можно судить о сходстве двух видов животных, двух заболеваний и т. д. Самая длинная общая подпоследовательность используется для поиска лекарства от рассеянного склероза.

- Вы когда-нибудь пользовались ключом `diff` (например, в команде `git diff`)? Этот ключ выводит информацию о различиях между двумя файлами, а для этого он использует динамическое программирование.
- Мы также упоминали о сходстве строк. *Расстояние Левенштейна* оценивает, насколько похожи две строки, а для его вычисления применяется динамическое программирование. Расстояние Левенштейна используется в самых разных областях, от проверки орфографии до выявления отправки пользователем данных, защищенных авторским правом.
- Вы когда-нибудь работали в приложении, поддерживающем перенос слов, например Microsoft Word? Как определить, где следует расставить переносы, чтобы длина строки оставалась более или менее постоянной? Динамическое программирование!

Упражнения

9.3 Нарисуйте и заполните таблицу для вычисления самой длинной общей подстроки между строками *blue* и *clues*.

Шпаргалка

- Динамическое программирование применяется при оптимизации некоторой характеристики.
- Динамическое программирование работает только в ситуациях, в которых задача может быть разбита на автономные подзадачи.
- В каждом решении из области динамического программирования строится таблица.
- Значения ячеек таблицы обычно соответствуют оптимизируемой характеристике.
- Каждая ячейка представляет подзадачу, поэтому вы должны подумать о том, как разбить задачу на подзадачи.
- Не существует единой формулы для вычисления решений методом динамического программирования.

10 Алгоритм *k* ближайших соседей

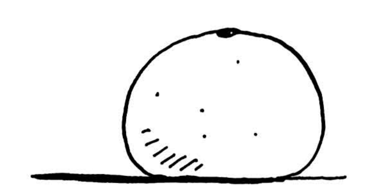


В этой главе

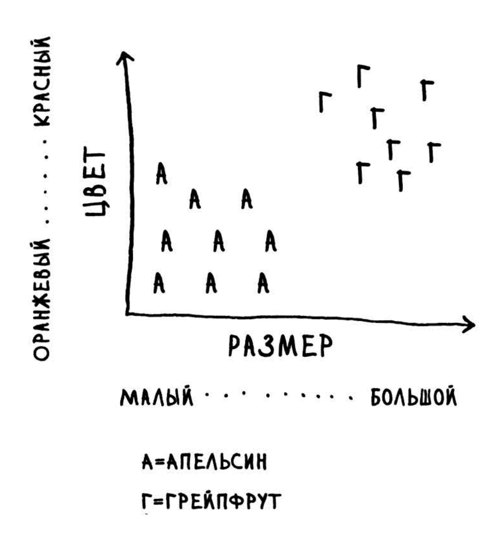
- ✓ Вы научитесь строить системы классификации на базе алгоритма *k* ближайших соседей.
- ✓ Вы узнаете об извлечении признаков.
- ✓ Вы узнаете о регрессии: прогнозировании чисел (например, завтрашних биржевых котировок или успеха фильма у зрителей).
- ✓ Вы познакомитесь с типичными сценариями использования и ограничениями алгоритма k ближайших соседей.

Апельсины и грейпфруты

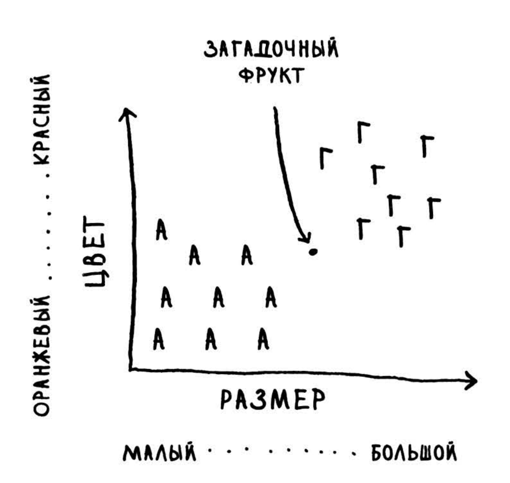
Взгляните на этот фрукт. Что это, апельсин или грейпфрут? Я слышал, что грейпфруты обычно крупнее, а их кожура имеет красноватый оттенок.



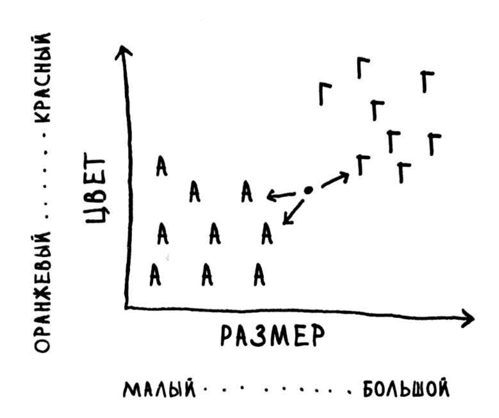
Мой мыслительный процесс выглядит примерно так: у меня в мозге существует некое подобие графика.



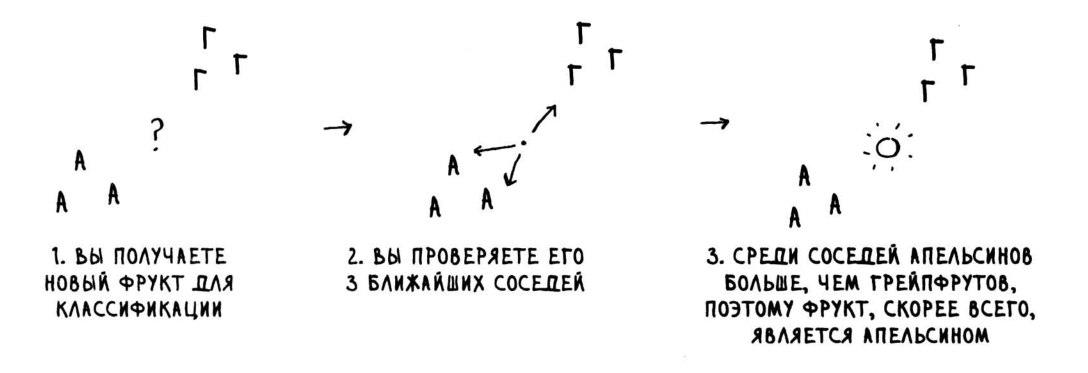
Как правило, крупные и красные фрукты оказываются грейпфрутами. Этот фрукт большой и красный, поэтому, скорее всего, это грейпфрут. Но что, если вам попадется фрукт вроде такого?



Как *классифицировать* этот фрукт? Один из способов — рассмотреть соседей этой точки. Возьмем ее трех ближайших соседей.



Среди соседей больше апельсинов, чем грейпфрутов. Следовательно, этот фрукт, скорее всего, является апельсином. Поздравляем: вы только что применили алгоритм ***k* ближайших соседей** для ***классификации***! В целом алгоритм работает по довольно простому принципу.

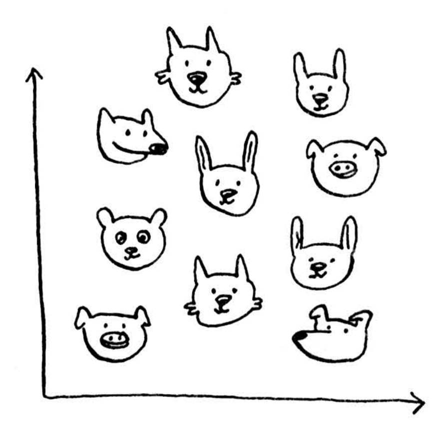


Алгоритм k ближайших соседей прост, но полезен! Если вы пытаетесь выполнить классификацию чего-либо, сначала попробуйте применить алгоритм k ближайших соседей. Рассмотрим более реалистичный пример.

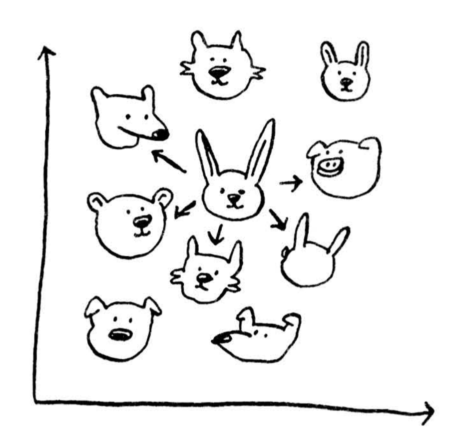
Построение рекомендательной системы

Представьте, что вы работаете на сайте Netflix и хотите построить систему, которая будет рекомендовать фильмы для ваших пользователей. На высоком уровне эта задача похожа на задачу с грейпфрутами!

Информация о каждом пользователе наносится на график.



Положение пользователя определяется его вкусами, поэтому пользователи с похожими вкусами располагаются недалеко друг от друга. Предположим, вы хотите порекомендовать фильмы Приянке. Найдите пять пользователей, ближайших к ней.



У Джастина, Джей-Си, Джозефа, Ланса и Криса похожие вкусы. Значит, те фильмы, которые нравятся *им*, с большой вероятностью понравятся и Приянке!

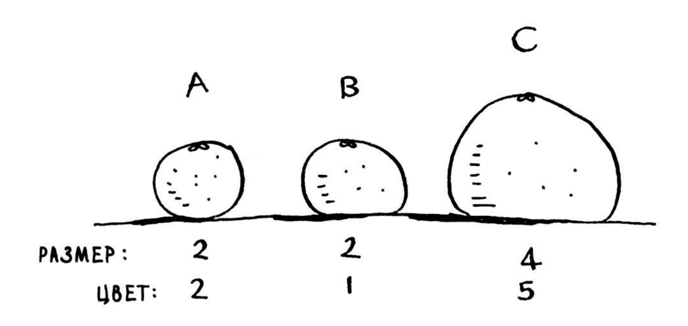
После того как у вас появится такая диаграмма, построить рекомендательную систему будет несложно. Если Джастину нравится какой-нибудь фильм, порекомендуйте этот фильм Приянке.



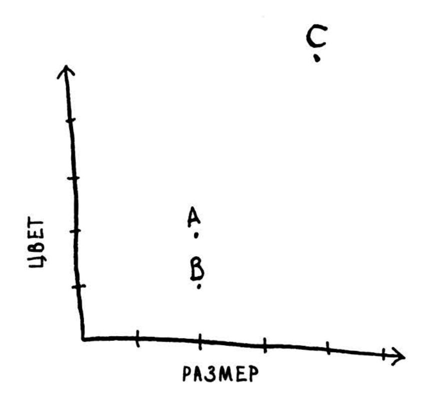
Однако в картине не хватает одного важного фрагмента. Вы оценивали, насколько близки вкусы двух пользователей на графике. Но как определить, насколько они близки?

Извлечение признаков

В примере с грейпфрутами мы сравнивали фрукты на основании их размера и цвета кожуры. Размер и цвет — *признаки*, по которым ведется сравнение. Теперь предположим, что у вас есть три фрукта. Вы можете извлечь из них информацию, то есть провести извлечение признаков.



Данные трех фруктов наносятся на график.



Из диаграммы хорошо видно, что фрукты A и B похожи. Давайте измерим степень их сходства. Для вычисления расстояния между двумя точками применяется формула Пифагора.

$$\sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

Например, расстояние между A и B вычисляется так:

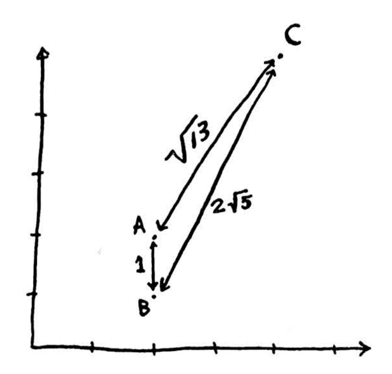
$$\sqrt{(2-2)^2 + (2-1)^2}$$

$$= \sqrt{0 + 1}$$

$$= \sqrt{1}$$

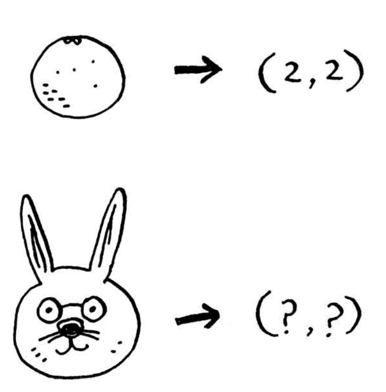
$$= 1$$

Расстояние между A и B равно 1. Другие расстояния вычисляются аналогично.



Формула расстояния подтверждает то, что мы видим: между фруктами A и B есть сходство.

Допустим, вместо фруктов вы сравниваете пользователей Netflix. Пользователей нужно будет как-то нанести на график. Следовательно, каждого пользователя нужно будет преобразовать в координаты — так же, как это было сделано для фруктов.



Когда вы сможете нанести пользователей на график, вы также сможете измерить расстояние между ними.

Начнем с преобразования пользователей в набор чисел. Когда пользователь регистрируется на Netflix, предложите ему оценить несколько категорий

фильмов: нравятся они лично ему или нет. Таким образом у вас появляется набор оценок для каждого пользователя!

	ПРИЯНКА	ДЖАСТИН	МОРФЕУС
КОМЕДИЯ	3	4	2
БОЕВИК	4	3	5
ДРАМА	4	5	1
УЖАСЫ	1	1	3
МЕЛОДРАМА	4	5	1

Приянка и Джастин обожают мелодрамы и терпеть не могут ужасы. Морфеусу нравятся боевики, но он не любит мелодрамы (хороший боевик не должен прерываться слащавой романтической сценой). Помните, как в задаче об апельсинах и грейпфрутах каждый фрукт представлялся двумя числами? Здесь каждый пользователь представляется набором из пяти чисел.

→ (2, 2)

→ (3, 4, 4, 1, 4)

Математик скажет, что вместо вычисления расстояния в двух измерениях вы теперь вычисляете расстояние в пяти измерениях. Тем не менее формула расстояния остается неизменной.

$$\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + (d_1 - d_2)^2 + (e_1 - e_2)^2}$$

Просто на этот раз используется набор из пяти чисел вместо двух.

Формула расстояния универсальна: даже если вы используете набор из миллиона чисел, расстояние вычисляется по той же формуле. Естественно спросить: какой смысл передает метрика *расстояния* с пятью числами? Она сообщает, насколько близки между собой эти наборы из пяти чисел.

$$\sqrt{(3-4)^2+(4-3)^2+(4-5)^2+(1-1)^2+(4-5)^2}$$
$$=\sqrt{1+1+1+0+1}$$
$$=\sqrt{4}$$
$$=2$$

Это расстояние между Приянкой и Джастином.

Вкусы Приянки и Джастина похожи. А насколько различаются вкусы Приянки и Морфеуса? Вычислите расстояние между ними, прежде чем продолжить чтение.

Сколько у вас получилось? Приянка и Морфеус находятся на расстоянии 24. По этому расстоянию можно понять, что у Приянки больше общего с Джастином, чем с Морфеусом.

Прекрасно! Теперь порекомендовать фильм Приянке будет несложно: если Джастину понравился какой-то фильм, мы рекомендуем его Приянке, и наоборот. Вы только что построили систему, рекомендующую фильмы.

Если вы являетесь пользователем Netflix, то Netflix постоянно напоминает вам: «Пожалуйста, оценивайте больше фильмов. Чем больше фильмов вы оцените, тем точнее будут наши рекомендации». Теперь вы знаете почему: чем больше фильмов вы оцениваете, тем точнее Netflix определяет, с какими пользователями у вас общие вкусы.

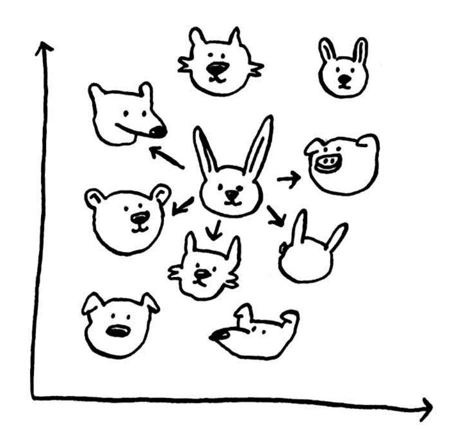
Упражнения

10.1 В примере с Netflix сходство между двумя пользователями оценивалось по формуле расстояния. Но не все пользователи оценивают фильмы одинаково. Допустим, есть два пользователя, Йоги и Пинки, вкусы которых совпадают. Но Йоги ставит 5 баллов любому фильму, который ему понравился, а Пинки более разборчива и ставит «пятерки» только самым лучшим фильмам. Вроде бы вкусы одинаковые, но по метрике расстояния они не являются соседями. Как учесть различия в стратегиях выставления оценок?

10.2 Предположим, Netflix определяет группу «авторитетов». Скажем, Квентин Тарантино и Уэс Андерсон относятся к числу авторитетов Netflix, поэтому их оценки оказывают более сильное влияние, чем оценки рядовых пользователей. Как изменить систему рекомендаций, чтобы она учитывала повышенную ценность оценок авторитетов?

Регрессия

А теперь предположим, что просто порекомендовать фильм недостаточно: вы хотите спрогнозировать, какую оценку Приянка поставит фильму. Возьмите 5 пользователей, находящихся вблизи от нее.



Кстати, я уже не в первый раз говорю о «ближайших пяти». В числе «5» нет ничего особенного: с таким же успехом можно взять 2 ближайших пользователей, 10 или 10 000. Поэтому-то алгоритм и называется «алгоритмом *k* ближайших пользователей», а не «алгоритмом 5 ближайших пользователей»!

Допустим, вы пытаетесь угадать оценку Приянки для фильма «Идеальный голос». Как этот фильм оценили Джастин, Джей-Си, Джозеф, Ланс и Крис?

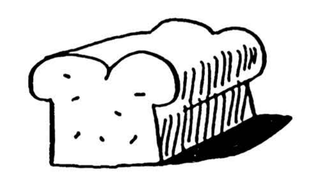
ДЖАСТИН : 5
ДЖЕЙ-СИ : 4
ДЖОЗЕФ : 4
ЛАНС : 5
КРИС : 3

Если вычислить среднее арифметическое их оценок, вы получите 4,2. Такой метод прогнозирования называется *регрессией*. У алгоритма *k* ближайших соседей есть два основных применения: классификация и регрессия:

- классификация = распределение по категориям;
- регресия = прогнозирование ответа (в числовом выражении).

Регрессия чрезвычайно полезна. Представьте, что вы открыли маленькую булочную в Беркли и каждый день выпекаете свежий хлеб. Вы пытаетесь предсказать, сколько буханок следует испечь на сегодня. Есть несколько признаков:

- погода по шкале от 1 до 5 (1 = плохая, 5 = отличная);
- праздник или выходной? (1, если сегодня праздник или выходной, 0 в противном случае);
- проходят ли сегодня спортивные игры? (1 = да, 0 = нет).



И вы знаете, сколько буханок хлеба было продано в прошлом при разных сочетаниях признаков.

A. (5, 1, 0) = 300 БУХАНОК
B. (3, 1, 1) = 225 БУХАНОК
C. (1, 1, 0) = 75 БУХАНОК
D. (4, 0, 1) = 200 БУХАНОК
E. (4, 0, 0) = 150 БУХАНОК
F. (2, 0, 0) = 50 БУХАНОК

Сегодня выходной и хорошая погода. Сколько буханок вы продадите на основании только что приведенных данных? Используем алгоритм *k* ближайших соседей для *k* = 4. Сначала определим четырех ближайших соседей для этой точки.

(4, 1, 0) = ?

Ниже перечислены расстояния. Точки A, B, D и E являются ближайшими.

A. 1 ←
B. 2 ←
C. 9
D. 2 ←
E. 1 ←
F. 5

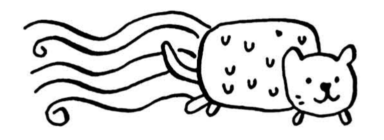
Вычисляя среднее арифметическое продаж в эти дни, вы получаете 218,75. Значит, именно столько буханок нужно выпекать на сегодня!

БЛИЗОСТЬ КОСИНУСОВ

До сих пор мы использовали формулу расстояния для вычисления степени сходства двух пользователей. Но является ли эта формула лучшей? На практике также часто применяется метрика близости косинусов. Допустим, два пользователя похожи, но один из них более консервативен в своих оценках. Обоим пользователям понравился фильм Манмохана Десаи «Амар Акбар Антони». Пол поставил фильму оценку 5 звезд, но Роуэн оценил его только в 4 звезды. Если использовать формулу расстояния, эти два пользователя могут не оказаться соседями, несмотря на сходство вкусов.

Метрика близости косинусов не измеряет расстояние между двумя векторами. Вместо этого она сравнивает углы двух векторов и в целом лучше подходит для подобных случаев. Тема метрики близости косинусов выходит за рамки этой книги, но вам стоит самостоятельно поискать информацию о ней, если вы будете применять алгоритм *k* ближайших соседей!

Выбор признаков



Чтобы подобрать рекомендации, вы предлагаете пользователям ставить оценки категориям фильмов. А если бы вы вместо этого предлагали им ставить оценки картинкам с котами? Наверное, вам бы удалось найти пользователей, которые ставили похожие оценки этим картинкам. Однако у вас получилась бы самая плохая рекомендательная система в мире, потому что эти «признаки» не имеют никакого отношения к их вкусам в области кино!

Или представьте, что вы предлагаете пользователям оценить фильмы для формирования рекомендаций — но только «Историю игрушек», «Историю игрушек-2» и «Историю игрушек-3». Эти оценки ничего не скажут вам о вкусах пользователей.

Когда вы работаете с алгоритмом *k* ближайших соседей, очень важно правильно выбрать признаки для сравнения. Под правильным выбором признаков следует понимать:

- признаки, напрямую связанные с фильмами, которые вы пытаетесь рекомендовать;
- признаки, не содержащие смещения (например, если предлагать пользователям оценивать только комедии, вы не получите никакой информации об их отношении к боевикам).

Как вы думаете, оценки хорошо подходят для рекомендации фильмов? Возможно, я поставил «Прослушке» более высокую оценку, чем «Охотникам за недвижимостью», но на самом деле я провел больше времени за просмотром «Охотников». Как улучшить рекомендательную систему Netflix?

Возвращаясь к примеру с пекарней: сможете ли вы придумать два хороших и два плохих признака, которые можно было бы выбрать для прогнозирования объема выпечки? Возможно, нужно выпечь побольше хлеба после рекламы в газете. Или увеличить объем производства по понедельникам.

В том, что касается выбора хороших признаков, не существует единственно правильного ответа. Тщательно продумайте все факторы, которые необходимо учесть при прогнозировании.

Упражнения

10.3 У сервиса Netflix миллионы пользователей. В приведенном ранее примере рекомендательная система строилась для пяти ближайших соседей. Пять — это слишком мало? Слишком много?

Знакомство с машинным обучением

Мало того, что алгоритм *k* ближайших соседей полезен — он открывает путь в волшебный мир машинного обучения! Суть машинного обучения — сделать ваш компьютер более разумным. Вы уже видели один пример машинного обучения: построение рекомендательной системы. В этом разделе будут рассмотрены другие примеры.

OCR

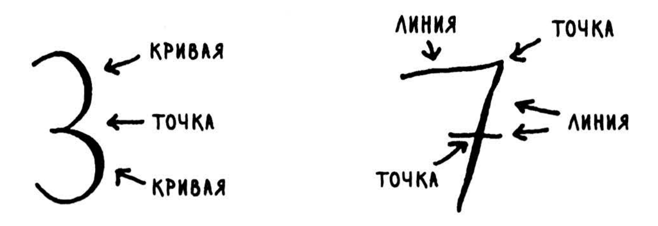
Сокращение OCR означает «Optical Character Recognition», то есть «оптическое распознавание текста». Иначе говоря, вы берете фотографию страницы текста, а компьютер автоматически преобразует изображение в текст. Google использует OCR для оцифровки книг. Как работает OCR? Для примера возьмем следующую цифру:



Как автоматически определить, что это за цифра? Можно воспользоваться алгоритмом k ближайших соседей:

1. Переберите изображения цифр и извлеките признаки.
2. Получив новое изображение, извлеките признаки и проверьте ближайших соседей.

По сути это та же задача, что и задача классификации апельсинов и грейпфрутов. В общем случае алгоритмы OCR основаны на выделении линий, точек и кривых.



Затем при получении нового символа из него можно извлечь те же признаки.

Извлечение признаков в OCR происходит намного сложнее, чем в примере с фруктами. Однако важно понимать, что даже сложные технологии строятся на основе простых идей (таких, как алгоритм *k* ближайших соседей).

Те же принципы могут использоваться для распознавания речи или распознавания лиц. Когда вы отправляете фотографию на Facebook, иногда сайту хватает сообразительности для автоматической пометки людей на фото. Да это машинное обучение в действии!

Первый шаг OCR, в ходе которого перебираются изображения цифр и происходит извлечение признаков, называется *тренировкой*. В большинстве алгоритмов машинного обучения присутствует фаза тренировки: прежде чем компьютер сможет решить свою задачу, его необходимо натренировать. В следующем примере рассматривается создание спам-фильтров, и в нем тоже есть шаг тренировки.

Построение спам-фильтра

Спам-фильтры используют другой простой алгоритм, называемый *наивным классификатором Байеса*. Сначала наивный классификатор Байеса тренируется на данных.

ТЕМА	СПАМ?
«ИЗМЕНИТЕ ПАРОЛЬ»	НЕ СПАМ
«ВЫ ВЫИГРАЛИ МИЛЛИОН»	СПАМ
«СООБЩИТЕ СВОЙ ПАРОЛЬ»	СПАМ
«НИГЕРИЙСКИЙ ПРИНЦ ГОТОВ ПЕРЕВЕСТИ ВАМ МИЛЛИОН»	СПАМ
«С ДНЕМ РОЖДЕНИЯ!»	НЕ СПАМ

Предположим, вы получили сообщение с темой «Получите свой миллион прямо сейчас!» Это спам? Предложение можно разбить на слова, а затем для каждого слова проверить вероятность присутствия этого слова в спамовом сообщении. Например, в нашей очень простой модели слово «*миллион*» встречается только в спаме. Наивный классификатор Байеса вычисляет вероятность того, что сообщение с большой вероятностью является спамом. На практике он применяется примерно для тех же целей, что и алгоритм k ближайших соседей.

Например, наивный классификатор Байеса может использоваться для классификации фруктов: есть большой и красный фрукт. Какова вероятность того, что он окажется грейпфрутом? Это простой, но весьма эффективный алгоритм — из тех, что нам нравятся больше всего!

Прогнозы на биржевых торгах

Есть одна задача, в которой трудно добиться успеха машинным обучением: точно спрогнозировать курсы акций на бирже. Как выбрать хорошие признаки? Предположим, вы говорите, что если курс акций рос вчера, то он будет расти и сегодня. Хороший это признак или нет? Или, предположим, вы утверждаете, что курс всегда снижается в мае. Сработает или нет? Не существует гарантированного способа прогнозировать будущее на основании прошлых данных. Прогнозирование будущего — сложное дело, а при таком количестве переменных оно становится почти невозможным.



Шпаргалка

Надеюсь, вы хотя бы в общих чертах поняли, что можно сделать с помощью алгоритма *k* ближайших соседей и машинного обучения! Машинное обучение — интересная область, и при желании в нее можно зайти достаточно глубоко.

- Алгоритм *k* ближайших соседей применяется для классификации и регрессии. В нем используется проверка *k* ближайших соседей.

- Классификация = распределение по категориям.
- Регрессия = прогнозирование результата (например, в виде числа).
- «Извлечением признаков» называется преобразование элемента (например, фрукта или пользователя) в список чисел, которые могут использоваться для сравнения.
- Качественный выбор признаков — важная часть успешного алгоритма *k* ближайших соседей.

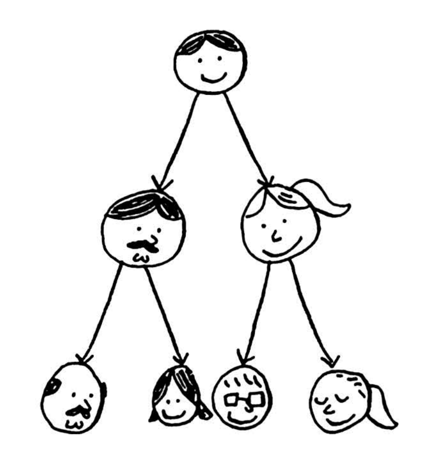
11 Что дальше?



В этой главе

- ✓ Приводится краткий обзор 10 алгоритмов, которые не рассматривались в книге. Вы узнаете, для чего нужны эти алгоритмы.
- ✓ Я порекомендую книги, которые стоит читать дальше в зависимости от того, какие темы представляют интерес для вас.

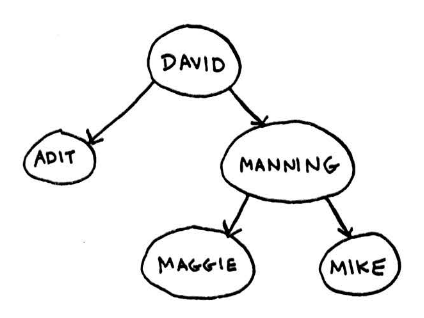
Деревья



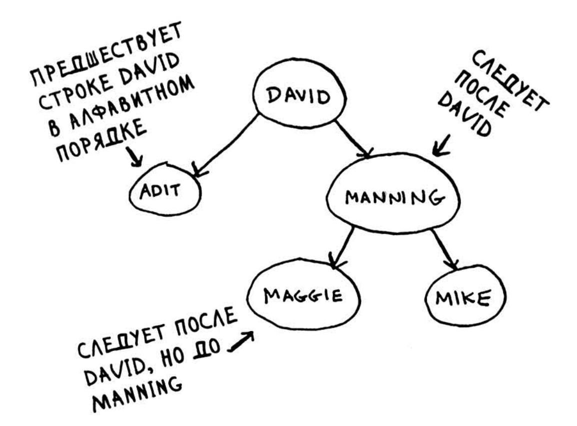
Вернемся к примеру с бинарным поиском. Когда пользователь вводит свое имя на сайте Facebook, сайт должен проверить содержимое большого массива, чтобы узнать, существует ли пользователь с таким именем. Мы выяснили, что для нахождения значения в массиве быстрее всего воспользоваться бинарным поиском. Однако здесь

возникает проблема: каждый раз, когда на сайте регистрируется новый пользователь, придется заново сортировать массив, потому что бинарный поиск работает только с отсортированными массивами. Насколько удобнее было бы вставить пользователя в правильную ячейку массива, чтобы потом его не пришлось сортировать заново! Именно эта идея заложена в основу структуры данных *бинарного дерева поиска*.

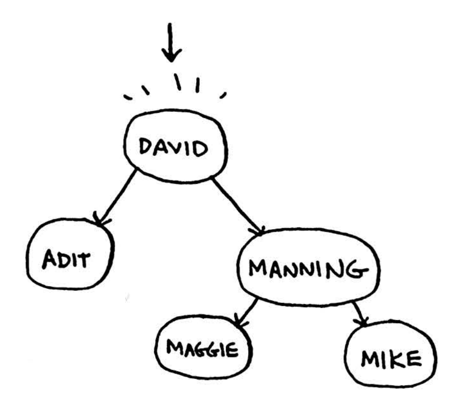
Бинарное дерево поиска выглядит так:



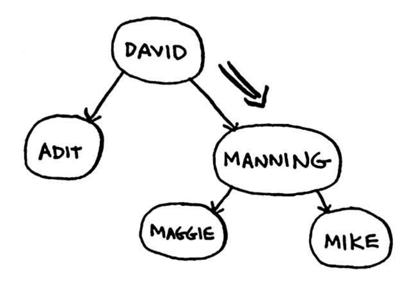
Для каждого узла все узлы левого поддерева содержат *меньшие* значения, а все узлы правого поддерева — *большие* значения.



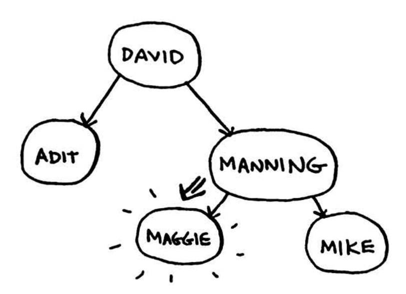
Предположим, вы ищете узел *Maggie*. Поиск начинается с корневого узла.



Строка *Maggie* идет после *David*, поэтому идем направо.



Строка *Maggie* предшествует *Manning*, поэтому идем налево.

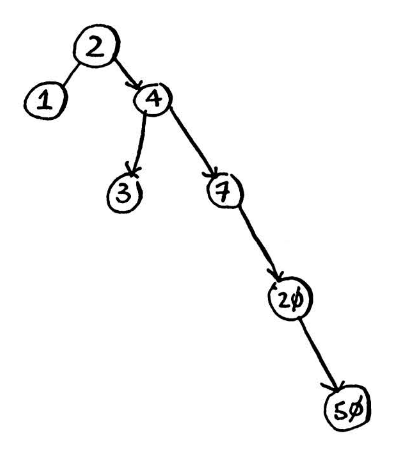


Мы нашли узел *Maggie*! В целом процедура поиска напоминает бинарный поиск. Поиск элемента в бинарном дереве поиска *в среднем* выполняется

за время $O(\log n)$, а в *худшем случае* — за время $O(n)$. Поиск в отсортированном массиве выполняется за время $O(\log n)$ *в худшем случае* — казалось бы, отсортированный массив эффективнее. Однако бинарное дерево поиска в среднем работает намного быстрее при удалении и вставке элементов.

	МАССИВ	БИНАРНОЕ ДЕРЕВО ПОИСКА
ПОИСК	$O(\log n)$	$O(\log n)$
ВСТАВКА	$O(n)$	$O(\log n)$
УДАЛЕНИЕ	$O(n)$	$O(\log n)$

У бинарных деревьев поиска есть и свои недостатки: во-первых, они не поддерживают произвольный доступ. Вы не сможете потребовать: «Выдайте мне i-й элемент этого дерева». Кроме того, в таблице приведено *среднее* время выполнения операций; оно зависит от сбалансированности дерева. Допустим, ваше дерево не сбалансировано, как на следующем рисунке.



Видите, как дерево перекошено вправо? Эффективность такого дерева оставляет желать лучшего, потому что это дерево не сбалансировано. Существуют специальные бинарные деревья поиска, способные к самобалансировке (как, например, красно-черные деревья).

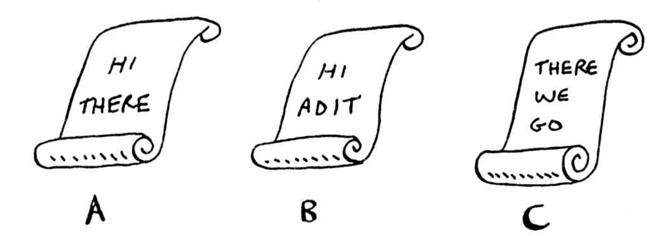
Где же используются бинарные деревья поиска? В-деревья, особая разновидность бинарных деревьев, обычно используются для хранения информации в базах данных.

Если вас интересуют базы данных или более сложные структуры данных, поищите информацию по следующим темам:

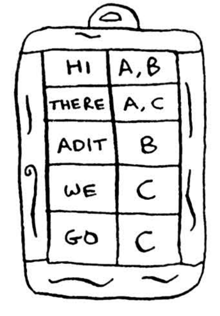
- в-деревья;
- красно-черные деревья;
- кучи;
- скошенные (splay) деревья.

Инвертированные индексы

Перед вами сильно упрощенное объяснение того, как работает поисковая система. Допустим, имеются три веб-страницы с простым содержимым.



Построим хеш-таблицу для этого содержимого.



Ключами хеш-таблицы являются слова, а значения указывают, на каких страницах встречается каждое слово. Теперь предположим, что пользователь ищет слово *hi*. Посмотрим, на каких страницах это слово встречается.



Ага, слово встречается на страницах A и B. Выведем эти страницы в результатах поиска. Или предположим, что пользователь ищет слово *there*. Вы знаете, что это слово встречается на страницах A и C. Несложно, верно? Это очень полезная структура данных: хеш-таблица, связывающая слова с местами, в которых эти слова встречаются. Такая структура данных, называемая ***инвертированным индексом***, часто используется для построения поисковых систем. Если вас интересует область поиска, эта тема станет хорошей отправной точкой для дальнейшего изучения.

Преобразование Фурье

Преобразование Фурье — действительно выдающийся алгоритм: великолепный, элегантный и имеющий миллион практических применений. Лучшая аналогия для преобразования Фурье приводится на сайте Better Explained (отличный веб-сайт, на котором просто объясняется математическая теория): если у вас есть коктейль, преобразование Фурье сообщает, из каких ингредиентов он состоит[1]. Или для заданной песни преобразование разделяет ее на отдельные частоты.

Оказывается, эта простая идея находит множество практических применений. Например, если песню можно разложить на частоты, вы можете усилить тот диапазон, который вас интересует, — скажем, усилить низкие частоты и приглушить высокие. Преобразование Фурье прекрасно подходит для обработки сигналов. Также оно может применяться для сжатия музыки: сначала звуковой файл разбивается на составляющие. Преобразование Фурье сообщает, какой вклад вносит каждая составляющая в музыку, что позволяет исключить несущественные составляющие. Собственно, именно так работает музыкальный формат MP3!

Музыка — не единственный вид цифровых сигналов. Графический формат JPG также использует сжатие и работает по тому же принципу. Преобразование Фурье также применяется для прогнозирования землетрясений и анализа ДНК.

С его помощью можно построить аналог Shazam — приложение, которое находит песни по отрывкам. Преобразование Фурье очень часто применяется на практике. Почти наверняка вы с ним еще столкнетесь!

---

[1] Kalid, «An Interactive Guide to the Fourier Transform,» Better Explained, http://mng.bx/874X.

Параллельные алгоритмы

Следующие три темы связаны с масштабируемостью и обработкой больших объемов данных. Когда-то компьютеры становились все быстрее и быстрее. Если вы хотели, чтобы ваш алгоритм работал быстрее, можно было подождать несколько месяцев и запустить программу на более мощном компьютере. Но сейчас этот период подошел к концу. Современные компьютеры и ноутбуки оснащаются многоядерными процессорами. Чтобы алгоритм заработал быстрее, необходимо преобразовать его в форму, подходящую для параллельного выполнения сразу на всех ядрах!

Рассмотрим простой пример. Лучшее время выполнения для алгоритма сортировки равно приблизительно $O(n \log n)$. Известно, что массив невозможно отсортировать за время $O(n)$, *если только не воспользоваться параллельным алгоритмом*! Существует параллельная версия быстрой сортировки, которая сортирует массив за время $O(n)$.

Параллельный алгоритм трудно разработать. И так же трудно убедиться в том, что он работает правильно, и понять, какой прирост скорости он обеспечивает. Одно можно заявить твердо: выигрыш по времени не линеен. Следовательно, если процессор вашего компьютера имеет два ядра вместо одного, из этого не следует, что ваш алгоритм по волшебству заработает вдвое быстрее. Это объясняется несколькими причинами.

- *Затраты ресурсов на управление параллелизмом* — допустим, нужно отсортировать массив из 1000 элементов. Как разбить эту задачу для выполнения на двух ядрах? Выделить каждому ядру 500 элементов, а затем объединить два отсортированных массива в один большой отсортированный массив? Слияние двух массивов требует времени.

- *Распределение нагрузки* — допустим, необходимо выполнить 10 задач, и вы назначаете каждому ядру 5 задач. Однако ядру А достаются все простые задачи, поэтому оно выполняет свою работу за 10 секунд, тогда как ядро В справится со сложными задачами только за минуту. Это означает, что ядро А целых 50 секунд простаивает, пока ядро В выполняет всю работу! Как организовать равномерное распределение работы, чтобы оба ядра трудились с одинаковой интенсивностью?

Если вас интересует теоретическая сторона производительности и масштабируемости, возможно, параллельные алгоритмы — именно то, что вам нужно!

MapReduce

Одна разновидность параллельных алгоритмов в последнее время становится все более популярной: *распределенные алгоритмы*. Конечно, параллельный алгоритм удобно запустить на компьютере, если для его выполнения потребуется от двух до четырех ядер, а если нужны сотни ядер? Тогда алгоритм записывается так, чтобы он мог выполняться на множестве машин. Алгоритм MapReduce — известный представитель семейства распределенных алгоритмов. Для работы с ним можно воспользоваться популярной системой с открытым кодом Apache Hadoop.

Для чего нужны распределенные алгоритмы?

Предположим, имеется таблица с миллиардами или триллионами записей и вы хотите применить к ней сложный вопрос SQL. Выполнить его в MySQL не удастся, потому что MySQL начнет «тормозить» уже после нескольких миллиардов записей. Используйте MapReduce через Hadoop!

Или, предположим, вам нужно обработать длинный список заданий. Обработка каждого задания занимает 10 секунд, всего требует обработки 1 миллион заданий. Если выполнять эту работу на одном компьютере, она займет несколько месяцев! Если бы ее можно было выполнить на 100 машинах, работа завершилась бы за несколько дней.

Распределенные алгоритмы хорошо работают в тех ситуациях, когда вам нужно выполнить большой объем работы и вы хотите сократить время ее выполнения. В основе технологии MapReduce лежат две простые идеи: функция отображения `map` и функция свертки `reduce`.

Функция map

Функция `map` проста: она получает массив и применяет одну функцию к каждому элементу массива. Скажем, в следующем примере происходит удваивание каждого элемента в массиве:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> arr2 = map(lambda x: 2 * x, arr1)
[2, 4, 6, 8, 10]
```

Массив `arr2` теперь содержит значения `[2, 4, 6, 8, 10]` — все элементы `arr1` увеличились вдвое! Удвоение выполняется достаточно быстро. Но представьте, что выполнение применяемой функции требует больше времени. Взгляните на следующий псевдокод:

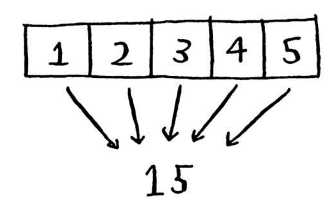
```
>>> arr1 = # Список URL
>>> arr2 = map(download_page, arr1)
```

Имеется список URL-адресов, нужно загрузить каждую страницу и сохранить содержимое в `arr2`. Для каждого адреса загрузка занимает пару секунд. Для 1000 адресов потребуется пара часов! А теперь представьте, что у вас имеется 100 машин и `map` автоматически распределяет работу между ними. Тогда в любой момент будут загружаться сразу 100 страниц одновременно, и работа пойдет намного быстрее!

Функция reduce

Функция `reduce` иногда сбивает людей с толку. Идея заключается в том, что весь список элементов «сокращается» до одного элемента. Напомню, что функция `map` переходит от одного массива к другому.

С функцией `reduce` массив преобразуется в один элемент.



Пример:

```
>>> arr1 = [1, 2, 3, 4, 5]
>>> reduce(lambda x,y: x+y, arr1)
15
```

В данном случае все элементы в массиве просто суммируются: 1 + 2 + 3 + 4 + 5 = 15! Я не буду рассматривать свертку более подробно, потому что в Интернете хватает руководств по этой теме.

MapReduce использует эти две простые концепции для выполнения запросов на нескольких машинах. При использовании большого набора данных (миллиарды записей) MapReduce выдаст ответ за минуты, тогда как традиционной базе данных на это потребуются многие часы.

Фильтры Блума и HyperLogLog

Представьте себя на месте сайта Reddit. Когда пользователь публикует ссылку, нужно проверить, публиковалась ли эта ссылка ранее. Истории, которые еще не публиковались, считаются более ценными.

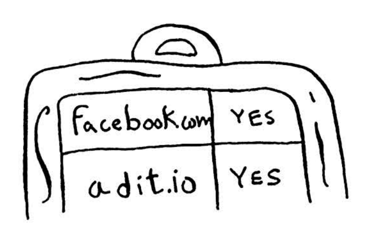
Или представьте себя на месте поискового бота Google. Обрабатывать веб-страницу нужно только в том случае, если она еще не обрабатывалась ранее. Итак, нужно проверить, обрабатывалась ли страница ранее.

Или представьте себя на месте *bit.ly* — сервиса сокращения URL. Пользователи не должны перенаправляться на вредоносные сайты. У вас имеется набор URL-адресов, которые считаются вредоносными. Теперь нужно выяснить, не направляется ли пользователь на URL-адрес из этого набора.

Во всех этих примерах возникает одна проблема. Имеется очень большой набор данных.



Появляется новый объект, и вы хотите узнать, содержится ли он в существующем наборе. Эта задача быстро решается при помощи хеша. Например, представьте, что Google создает большой хеш, ключами которого являются все обработанные страницы.



Как узнать, обрабатывался ли сайт *adit.io*? Нужно заглянуть в хеш.

adit.io → YES

У *adit.io* имеется свой ключ в хеше, а значит, адрес уже обрабатывался. Среднее время обращения к элементам в хеш-таблице составляет $O(1)$. Таким образом, вы узнали о том, что страница *adit.io* уже проиндексирована за постоянное время. Неплохо!

Вот только этот хеш получится просто *огромным*. Google индексирует триллионы веб-страниц. Если хеш содержит все URL-адреса, индексируемые Google, он займет слишком много места. У Reddit и *bit.ly* возникает аналогичная проблема. Сталкиваясь с такими объемами данных, приходится действовать более изобретательно!

Фильтры Блума

Для решения проблемы можно воспользоваться *вероятностными структурами данных*, которые называются *фильтрами Блума*. Они дают ответ, который может оказаться ложным, но с большой вероятностью является правильным. Вместо того чтобы обращаться к хешу, вы спрашиваете у фильтра Блума, обрабатывался ли этот URL-адрес ранее. Хеш-таблица даст точный ответ. Фильтр Блума дает ответ, правильный с высокой вероятностью:

- возможны ложно-положительные срабатывания. Фильтр скажет: «Этот сайт уже обрабатывался», хотя этого не было;
- ложно-отрицательные срабатывания исключены. Если фильтр утверждает, что сайт не обрабатывался, вы можете быть в этом уверены.

Фильтры Блума хороши тем, что занимают очень мало места. Хеш-таблице пришлось бы хранить все URL-адреса, обрабатываемые Google, а фильтру Блума это не нужно. Фильтры Блума очень удобны тогда, когда не нужно хранить точный ответ (как во всех приведенных примерах). Например, *bit.ly* может сказать: «Мы полагаем, что сайт может оказаться вредоносным, будьте особенно внимательны».

HyperLogLog

Примерно так же действует другой алгоритм, который называется HyperLogLog. Предположим, Google хочет подсчитать количество *уникальных* поисков, выполненных пользователями. Или Amazon хочет подсчитать количество уникальных предметов, просмотренных пользователями за сегодняшний день. Для получения ответов на эти вопросы потребуется очень много места! Так, в примере с Google придется вести журнал всех уникальных вариантов поиска. Когда пользователь что-то ищет, вы сначала

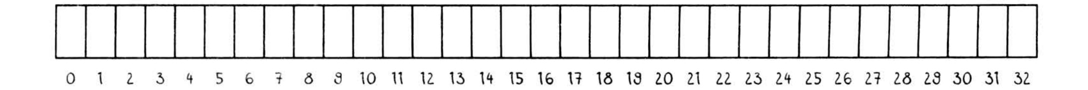
проверяете, присутствует ли условие в журнале, и если нет, добавляете его. Даже для одного дня этот журнал получится гигантским.

HyperLogLog аппроксимирует количество уникальных элементов в множестве. Как и фильтры Блума, он не дает точного ответа, но выдает достаточно близкий результат с использованием малой части памяти, которую обычно занимает такая задача.

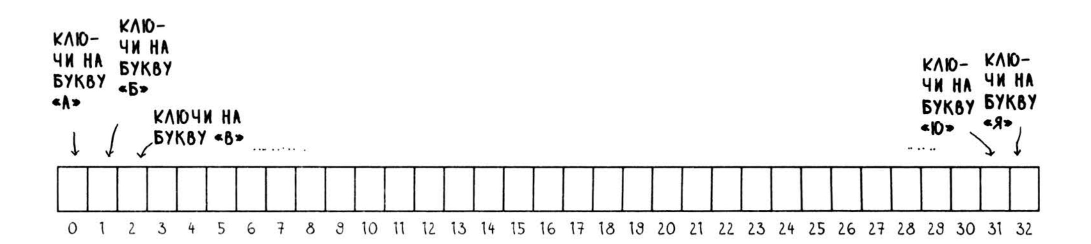
Если вы используете большие объемы данных и вас устраивают приближенные ответы — воспользуйтесь вероятностными алгоритмами!

Алгоритмы SHA

Помните процедуру хеширования из главы 5? На всякий случай освежу вашу память: имеется ключ, вы хотите поместить связанное с ним значение в массив.



Элемент, в котором размещается значение, определяется хеш-функцией.



Значение сохраняется в соответствующей позиции массива.



Хеширование позволяет выполнять поиск с постоянным временем. Когда вам потребуется узнать значение, связанное с ключом, вы снова применяете хеш-функцию, и она за время $O(1)$ сообщает, какую позицию следует проверить.

Хеш-функция должна обеспечивать достаточно равномерное распределение. Итак, хеш-функция получает строку и возвращает номер ячейки, соответствующий этой строке.

Сравнение файлов

Одну из разновидностей хеш-функций составляет алгоритм SHA (Secure Hash Algorithm). Он получает строку и возвращает хеш-код этой строки.

"hello" ⇒ 2cf24db...

Возможно, терминология не настолько проста, насколько хотелось бы. Алгоритм SHA — хеш-функция; эта функция генерирует хеш-код, который представляет собой короткую строку. Хеш-функция для хеш-таблиц преобразует строку в индекс массива, тогда как SHA преобразует строку в другую строку.

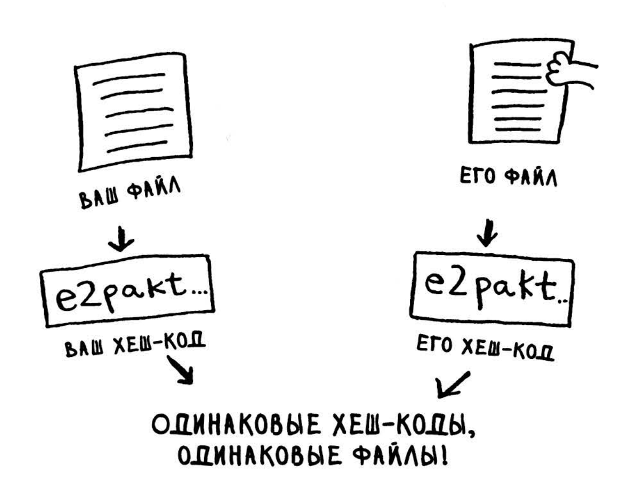
Для каждой строки алгоритм SHA генерирует свой уникальный хеш-код.

"hello" ⇒ 2cf24db...
"algorithm" ⇒ b1eb2ec...
"password" ⇒ 5e88489...

ПРИМЕЧАНИЕ

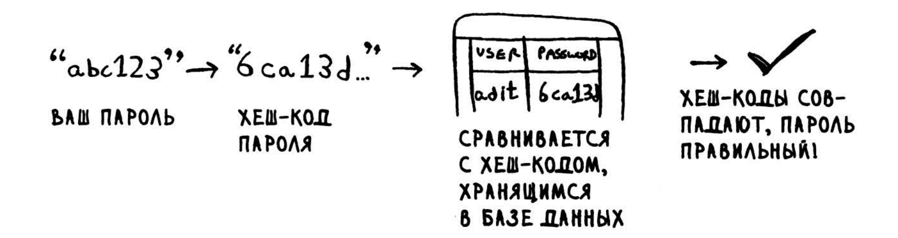
Хеш-коды SHA достаточно длинные. Здесь приводится только начало.

Алгоритм SHA позволяет определить, совпадают ли два файла. Такая возможность особенно полезна для очень больших файлов. Допустим, у вас имеется 4-гигабайтный файл и вы хотите проверить, хранится ли у вашего друга точно такой же файл. Вам не придется пересылать большой файл по электронной почте; вместо этого можно вычислить хеш-коды SHA двух файлов и сравнить их.



Проверка паролей

Алгоритм SHA также может использоваться для сравнения строк при отсутствии информации об исходной строке. Например, только представьте, что сервис Gmail атакован хакерами! Ваш пароль стал добычей злоумышленников? А вот и нет. Google хранит не исходный пароль, а только хеш-код пароля по алгоритму SHA! Когда вы вводите пароль, Google хеширует его и сравнивает результат с хеш-кодом, хранящимся в базе данных.



Сравниваются только хеш-коды — хранить пароль не нужно! Алгоритм SHA очень часто используются для хеширования паролей. Хеширование является односторонним: вы можете получить хеш-код строки...

abc123 → 6ca13d

...но не сможете восстановить исходную строку по хеш-коду:

? ← 6ca13d

Это означает, что даже если злоумышленник похитит хеш-коды SHA с серверов Gmail, он не сможет по ним восстановить исходные пароли! Пароль можно преобразовать в хеш, но не наоборот.

Под термином SHA скрывается целое семейство алгоритмов: SHA-0, SHA-1, SHA-2 и SHA-3. На момент написания книги в алгоритмах SHA-0 и SHA-1 были обнаружены слабости. Если вы применяете алгоритм SHA для хеширования паролей, выбирайте SHA-2 или SHA-3. В настоящее время «золотым стандартом» хеширования паролей считается функция bcrypt (хотя идеальной защиты не бывает).

Локально-чувствительное хеширование

У хеширования SHA есть еще одна важная особенность: оно является *локально-нечувствительным*. Предположим, имеется строка, для которой генерируется хеш-код:

dog → cd6357

Если изменить в строке всего один символ, а потом сгенерировать хеш заново, строка полностью изменяется!

dot → e392da

И это хорошо, потому что сравнение хешей не позволит атакующему определить, насколько он близок к взлому пароля.

Иногда требуется обратный результат: локально-чувствительная функция хеширования. Здесь на помощь приходит алгоритм *Simhash*. При незначительном изменении строки Simhash генерирует хеш-код, который почти не отличается от исходного. Это позволяет сравнивать хеш-коды и определять, насколько похожи две строки, — весьма полезная возможность!

- Google использует Simhash для выявления дубликатов в процессе индексирования.
- Преподаватель может использовать Simhash для обнаружения плагиата (копирования рефератов из Интернета).
- Scribd позволяет пользователям загружать документы или книги, чтобы они стали доступны для других пользователей. Но Scribd не хочет, чтобы пользователи размещали информацию, защищенную авторским правом! С помощью Simhash сайт может обнаружить, что отправленная информация похожа на книгу о Гарри Поттере, и при обнаружении сходства автоматически запретить ее размещение.

Simhash используется для выявления сходства между фрагментами текста.

Обмен ключами Диффи—Хеллмана

Алгоритм Диффи—Хеллмана заслуживает упоминания, потому что он изящно решает давно известную задачу. Как зашифровать сообщение так, чтобы его мог прочитать только тот человек, которому адресовано сообщение?

Проще всего определить подстановочный шифр: $a = 1$, $b = 2$ и т. д. Если после этого я отправлю вам сообщение «4,15,7», вы сможете преобразовать его в «d,o,g». Но чтобы эта схема сработала, необходимо согласовать шифр между сторонами. Договориться о шифре по электронной почте невозможно, потому что злоумышленник может перехватить сообщение, узнать шифр и расшифровать сообщения. Даже если передать шифр при личной встрече, злоумышленник может угадать шифр, если он достаточно прост. Значит, шифр придется ежедневно менять. Но тогда нам придется ежедневно проводить личные встречи для изменения шифра!

Даже если вам удастся ежедневно изменять шифр, подобные простые шифры достаточно легко взламываются методом грубой силы. Допустим, я вижу сообщение «9,6,13,13,16 24,16,19,13,5». Я предполагаю, что при шифровании используется подстановка $a = 1$, $b = 2$ и т. д.

9 6 13 13 16 24 16 19 13 5

I F M M P X P S M E

Бессмыслица. Пробуем $a = 2$, $b = 3$ и т. д.

9 6 13 13 16 24 16 19 13 5

H E L L O W O R L D

Сработало! Подобные простые шифры взламываются достаточно легко. Во Вторую мировую войну в Германии использовался намного более сложный шифр, но и он был взломан.

Алгоритм Диффи–Хеллмана решает обе проблемы:

- знание шифра обеими сторонами не обязательно. Следовательно, им не придется встречаться и согласовывать шифр;
- расшифровать зашифрованные сообщения *чрезвычайно* сложно.

Алгоритм Диффи–Хеллмана использует два ключа: открытый и закрытый. Открытый ключ известен обеим сторонам. Его можно опубликовать на сайте, отправить электронной почтой друзьям и вообще сделать с ним все, что вам заблагорассудится. Его не нужно скрывать. Когда другая сторона захочет отправить вам сообщение, она зашифрует его с применением открытого ключа. Зашифрованное сообщение можно расшифровать только с закрытым ключом. При условии, что вы являетесь единственным владельцем закрытого ключа, никто другой расшифровать сообщение не сможет!

Алгоритм Диффи—Хеллмана продолжает применяться на практике вместе с его наследником RSA. Если вы интересуетесь криптографией, алгоритм Диффи—Хеллмана станет хорошей отправной точкой: он элегантен и не особо сложен.

Линейное программирование

Самое лучшее я приберег напоследок. Линейное программирование — одна из самых интересных областей, которые мне известны.

Линейное программирование используется для максимизации некоторой характеристики при заданных ограничениях. Предположим, ваша компания выпускает два продукта: рубашки и сумки. На рубашку требуется 1 м ткани и 5 пуговиц. На изготовление сумки необходимо 2 м ткани и 2 пуговицы. У вас есть 11 м ткани и 20 пуговиц. Рубашка приносит прибыль $2, а сумка — $3. Сколько рубашек и сумок следует изготовить для получения максимальной прибыли?

Здесь мы пытаемся максимизировать прибыль, а ограничения определяют количество имеющихся материалов.

Другой пример: вы политик, пытающийся получить максимальное количество голосов. Исследования показали, что на каждый голос жителя Сан-Франциско требуется примерно час работы (маркетинг, исследования и т. д.), а на каждый голос жителя Чикаго — 1,5 часа. Вам нужны голоса как минимум 500 жителей Сан-Франциско и как минимум 300 жителей Чикаго. В вашем распоряжении 50 дней. Кроме того, затраты на жителя Сан-Франциско составляют $2, а на жителя Чикаго — $1. Ваш бюджет составляет $1500. Какое максимальное количество голосов вы сможете получить (Сан-Франциско+Чикаго)?

На этот раз вы стремитесь к максимуму голосов при ограничениях по времени и деньгам.

Возможно, вы думаете: «В этой книге много говорилось о вопросах оптимизации. Как они связаны с линейным программированием?» Все алгоритмы, работающие с графами, могут быть реализованы средствами линейного программирования. Линейное программирование — намного более общая область, а задачи с графами составляют ее подмножество.

В линейном программировании используется *симплекс-метод*. Этот алгоритм достаточно сложен, поэтому я не привожу его в книге. Если вы интересуетесь задачами оптимизации, поищите информацию о линейном программировании!

Эпилог

Надеюсь, этот краткий обзор показал, как много вам еще предстоит узнать. Я считаю, что лучший способ узнать что-то — найти тему, которая вас интересует, и изучить ее. Надеюсь, эта книга закладывает достаточно надежную основу для этого.

Ответы к упражнениям



Глава 1

1.1 Имеется отсортированный список из 128 имен, и вы ищете в нем значение методом бинарного поиска. Какое максимальное количество проверок для этого может потребоваться?

Ответ: 7

1.2 Предположим, размер списка увеличился вдвое. Как изменится максимальное количество проверок?

Ответ: 8

1.3 Известна фамилия, нужно найти номер в телефонной книге.

Ответ: $O(\log n)$

1.4 Известен номер, нужно найти фамилию в телефонной книге. (Подсказка: вам придется провести поиск по всей книге!)

Ответ: $O(n)$.

1.5 Нужно прочитать номера всех людей в телефонной книге.

Ответ: $O(n)$.

1.6 Нужно прочитать телефоны всех людей, фамилии которых начинаются с буквы «А». (Вопрос с подвохом! В нем задействованы концепции, которые более подробно рассматриваются в главе 4. Прочитайте ответ — скорее всего, он вас удивит!)

Ответ: $O(n)$. Возможно, кто-то подумает: «Я делаю это только для одной из 26 букв, а значит, время выполнения должно быть равно $O(n/26)$.» Запомните простое правило: в «O-большое» игнорируются числа, задействованные в операциях сложения, вычитания, умножения или деления. Ни одно из следующих значений не является правильной записью «O-большое»: $O(n + 26)$, $O(n - 26)$, $O(n * 26)$, $O(n / 26)$. Все они эквивалентны $O(n)$! Почему? Если вам интересно, найдите раздел «Снова об “O-большом”» в главе 4 и прочитайте о константах в этой записи (константа — это просто число; в этом вопросе 26 является константой).

Глава 2

2.1 Допустим, вы строите приложение для управления финансами.

1. ПРОДУКТЫ
2. КИНО
3. ВЕЛОСИПЕДНЫЙ КЛУБ

Ежедневно вы записываете все свои траты. В конце месяца вы анализируете расходы и вычисляете, сколько денег было потрачено. При работе с данными выполняется множество операций вставки и относительно немного операций чтения. Какую структуру использовать — массив или список?

Ответ: В данном случае траты добавляются в список ежедневно, а чтение всех данных происходит один раз в месяц. Для массивов характерно быстрое чтение и медленная вставка, а для связанных списков — медленное чтение и быстрая вставка. Так как вставка будет выполняться намного чаще, чем чтение, есть смысл воспользоваться связанным списком. Кроме того, чтение в связанных списках происходит медленно только при обращении к случайным элементам списка. Так как читаться будут все элементы списка, связанный список также неплохо справится с чтением. Итак, связанный список станет хорошим решением этой задачи.

2.2 Допустим, вы пишете приложение для приема заказов от посетителей ресторана. Приложение должно хранить список заказов. Официанты добавляют заказы в список, а повара читают заказы из списка и выполняют их. Заказы образуют очередь: официанты добавляют заказы в конец очереди, а повар берет первый заказ из очереди и начинает готовить.



Какую структуру данных вы бы использовали для реализации этой очереди — массив или связанный список? (Подсказка: связанные списки хорошо подходят для вставки/удаления, а массивы — для произвольного доступа к элементам. Что из этого понадобится в данном случае?)

Ответ: Связанный список. Вставка происходит очень часто (официанты добавляют заказы), а связанные списки эффективно выполняют эту операцию. Ни поиск, ни произвольный доступ (сильные стороны массивов) вам не понадобятся, потому что повар всегда извлекает из очереди первый заказ.

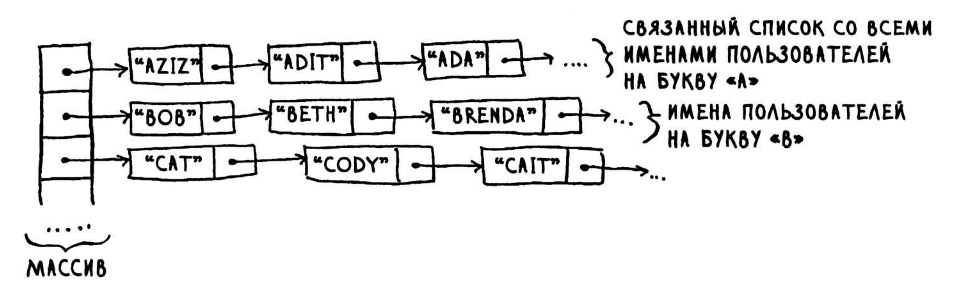
2.3 Проведем мысленный эксперимент. Допустим, Facebook хранит список имен пользователей. Когда кто-то пытается зайти на сайт Facebook, система пытается найти имя пользователя. Если имя входит в список имен зарегистрированных пользователей, то вход разрешается. Пользователи приходят на Facebook достаточно часто, поэтому поиск по списку имен пользователей будет выполняться часто. Будем считать, что Facebook использует бинарный поиск для поиска в списке. Бинарному поиску необходим произвольный доступ — алгоритм должен мгновенно обратиться к среднему элементу текущей части списка. Зная это обстоятельство, как бы вы реализовали список пользователей — в виде массива или связанного списка?

Ответ: В виде отсортированного массива. Массивы обеспечивают произвольный доступ — вы можете мгновенно получить элемент из середины массива. Со связанными списками это невозможно. Чтобы получить элемент из середины связанного списка, вам придется начать с первого элемента и переходить по ссылкам до нужного элемента.

2.4 Пользователи также довольно часто создают новые учетные записи на Facebook. Предположим, вы решили использовать массив для хранения списка пользователей. Какими недостатками обладает массив для выполнения вставки? Допустим, вы используете бинарный поиск для нахождения учетных данных. Что произойдет при добавлении новых пользователей в массив?

Ответ: Вставка в массив выполняется медленно. Кроме того, если вы используете бинарный поиск для нахождения имен пользователей, массив необходимо отсортировать. Предположим, пользователь по имени *Adit B* регистрируется на Facebook. Его имя будет вставлено в конец массива. Следовательно, массив нужно будет сортировать при каждой вставке нового имени!

2.5 В действительности Facebook не использует ни массив, ни связанный список для хранения информации о пользователях. Рассмотрим гибридную структуру данных: массив связанных списков. Имеется массив из 26 элементов. Каждый элемент содержит ссылку на связанный список. Например, первый элемент массива указывает на связанный список всех имен пользователей, начинающихся на букву «A». Второй элемент указывает на связанный список всех имен пользователей, начинающихся на букву «B», и т. д.



Предположим, пользователь с именем «Adit B» регистрируется в Facebook и вы хотите добавить его в список. Вы обращаетесь к элементу 1 массива, находите связанный список элемента 1 и добавляете

«Adit B» в конец списка. Теперь предположим, что зарегистрировать нужно пользователя «Zakhir H». Вы обращаетесь к элементу 26, который содержит связанный список всех имен, начинающихся с «Z», и проверяете, присутствует ли «Zakhir H» в этом списке.

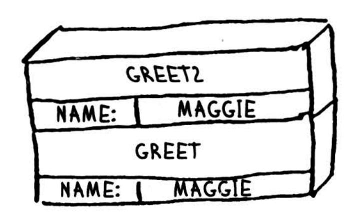
Теперь сравните эту гибридную структуру данных с массивами и связанными списками. Будет она быстрее или медленнее каждой исходной структуры при поиске и вставке? Приводить время выполнения «О-большое» не нужно, просто выберите одно из двух: быстрее или медленнее.

Ответ: Поиск — медленнее, чем для массивов, и быстрее, чем для связанных списков. Вставка — быстрее, чем для массивов, и с такой же скоростью для связанных списков. Итак, гибридная структура уступает массиву по скорости поиска, но по крайней мере не хуже связанных списков для всего остального. Далее в книге будет рассмотрена другая гибридная структура данных, называемая хеш-таблицей. Она даст некоторое представление о том, как строить сложные структуры данных из более простых.

Что же в действительности использует сервис Facebook? Вероятно, десяток разных баз данных, за которыми стоят разные структуры данных: хеш-таблицы, в-деревья и т. д. Массивы и связанные списки становятся структурными элементами для построения более сложных структур данных.

Глава 3

3.1 Предположим, имеется стек вызовов следующего вида:



Что можно сказать о текущем состоянии программы на основании этого стека вызовов?

Ответ: Некоторые наблюдения, о которых вы могли бы упомянуть:

- сначала вызывается функция `greet` для переменной `name = maggie`;
- затем функция `greet` вызывает функцию `greet2` для переменной `name = maggie`;
- на этой стадии функция `greet` находится в незавершенном, приостановленном состоянии;
- текущим вызовом функции является вызов `greet2`;
- после завершения этого вызова функция `greet` продолжит выполнение.

3.2 Предположим, вы случайно написали рекурсивную функцию, которая бесконечно вызывает саму себя. Как вы уже видели, компьютер выделяет память в стеке при каждом вызове функции. А что произойдет со стеком при бесконечном выполнении рекурсии?

Ответ: Стек будет расти бесконечно. Каждой программе выделяется ограниченный объем памяти в стеке. Когда все пространство будет исчерпано (а рано или поздно это произойдет), программа завершится с ошибкой переполнения стека.

Глава 4

4.1 Напишите код для функции `sum` (см. выше).

Ответ:

```
def sum(list):
  if list == []:
    return 0
  return list[0] + sum(list[1:])
```

4.2 Напишите рекурсивную функцию для подсчета элементов в списке.

Ответ:

```
def count(list):
  if list == []:
    return 0
  return 1 + count(list[1:])
```

4.3 Найдите наибольшее число в списке.

Ответ:

```
def max(list):
  if len(list) == 2:
    return list[0] if list[0] > list[1] else list[1]
  sub_max = max(list[1:])
  return list[0] if list[0] > sub_max else sub_max
```

4.4 Помните бинарный поиск из главы 1? Он тоже относится к классу алгоритмов «разделяй и властвуй». Сможете ли вы определить базовый и рекурсивный случай для бинарного поиска?

Ответ: Базовым случаем для бинарного поиска является массив, содержащий всего один элемент. Если искомый элемент совпадает с элементом массива – вы нашли его! В противном случае элемент в массиве отсутствует.

В рекурсивном случае для бинарного поиска массив делится пополам, одна половина отбрасывается, а для другой половины проводится бинарный поиск.

Запишите «О-большое» для каждой из следующих операций.

4.5 Вывод значения каждого элемента массива.

Ответ: $O(n)$.

4.6 Удвоение значения каждого элемента массива.

Ответ: $O(n)$.

4.7 Удвоение значения только первого элемента массива.

Ответ: $O(1)$.

4.8 Создание таблицы умножения для всех элементов массива. Например, если массив состоит из элементов [2, 3, 7, 8, 10], сначала каждый элемент умножается на 2, затем каждый элемент умножается на 3, затем на 7 и т. д.

Ответ: $O(n^2)$.

Глава 5

Какие из следующих функций являются последовательными?

5.1 `f(x) = 1` ← Возвращает "1" для любых входных значений

Ответ: Функция последовательна.

5.2 `f(x) = rand()` ← Возвращает случайное число

Ответ: Функция непоследовательна.

5.3 `f(x) = next_empty_slot()` ← Возвращает индекс следующего пустого элемента в хеш-таблице

Ответ: Функция непоследовательна.

5.4 `f(x) = len(x)` ← Возвращает длину полученной строки

Ответ: Функция последовательна.

Предположим, имеются четыре хеш-функции, которые получают строки.

1. Первая функция возвращает «1» для любого входного значения.
2. Вторая функция возвращает длину строки в качестве индекса.
3. Третья функция возвращает первый символ строки в качестве индекса. Таким образом, все строки, начинающиеся с «a», хешируются в одну позицию, все строки, начинающиеся с «b», — в другую и т. д.
4. Четвертая функция ставит в соответствие каждой букве простое число: a = 2, b = 3, c = 5, d = 7, e = 11 и т. д. Для строки хеш-функцией становится остаток от деления суммы всех значений на размер хеша. Например, если размер хеша равен 10, то для строки «bag» будет вычислен индекс 3 + 2 + 17 % 10 = 22 % 10 = 2.

В каком из этих примеров хеш-функции будут обеспечивать хорошее распределение? Считайте, что хеш-таблица содержит 10 элементов.

5.5 Телефонная книга, в которой ключами являются имена, а значениями — номера телефонов. Задан следующий список имен: Esther, Ben, Bob, Dan.

Ответ: Хеш-функции C и D обеспечивают хорошее распределение.

5.6 Связь размера батарейки с напряжением. Размеры батареек: A, AA, AAA, AAAA.

Ответ: Хеш-функции B и D обеспечивают хорошее распределение.

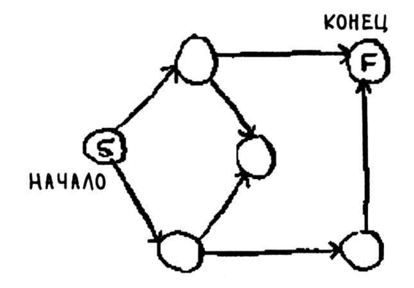
5.7 Связь названий книг с именами авторов. Названия книг: «Maus», «Fun Home», «Watchmen».

Ответ: Хеш-функции B, C и D обеспечивают хорошее распределение.

Глава 6

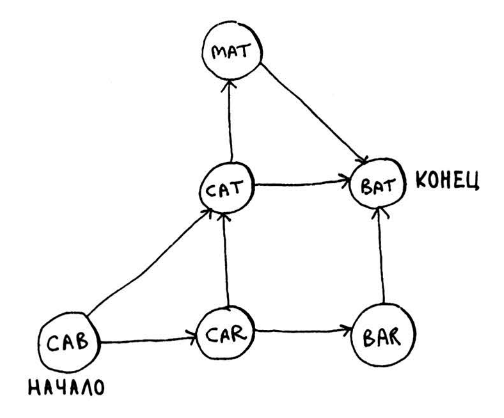
Примените алгоритм поиска в ширину к каждому из этих графов, чтобы найти решение.

6.1 Найдите длину кратчайшего пути от начального до конечного узла.



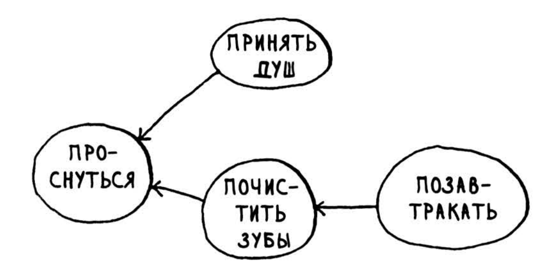
Ответ: Длина кратчайшего пути равна 2.

6.2 Найдите длину кратчайшего пути от «cab» к «bat».



Ответ: Длина кратчайшего пути равна 2.

6.3 Перед вами небольшой граф моего утреннего распорядка.

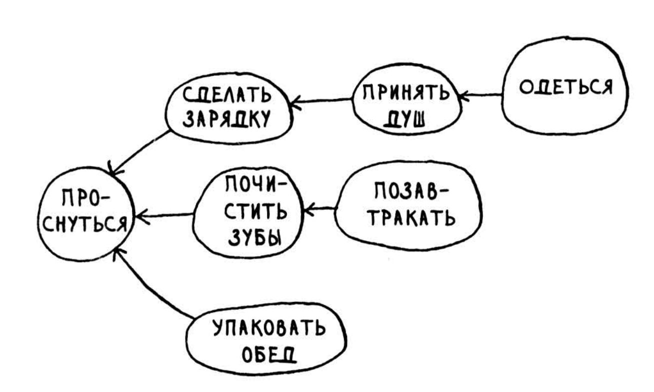


Для каждого из следующих трех списков укажите, действителен он или недействителен.

A.	B.	C.
1. ПРОСНУТЬСЯ	1. ПРОСНУТЬСЯ	1. ПРИНЯТЬ ДУШ
2. ПРИНЯТЬ ДУШ	2. ПОЧИСТИТЬ ЗУБЫ	2. ПРОСНУТЬСЯ
3. ПОЗАВТРАКАТЬ	3. ПОЗАВТРАКАТЬ	3. ПОЧИСТИТЬ ЗУБЫ
4. ПОЧИСТИТЬ ЗУБЫ	4. ПРИНЯТЬ ДУШ	4. ПОЗАВТРАКАТЬ

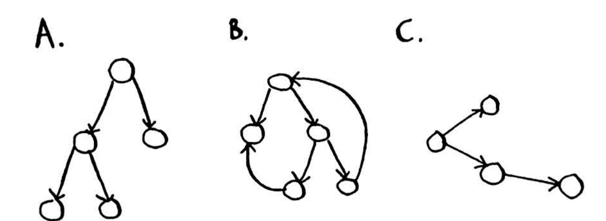
Ответы: A — недействителен; B — действителен; C — недействителен.

6.4 Немного увеличим исходный граф. Постройте действительный список для этого графа.



Ответ: 1 — Проснуться; 2 — Сделать зарядку; 3 — Принять душ; 4 — Почистить зубы; 5 — Одеться; 6 — Упаковать обед; 7 — Позавтракать.

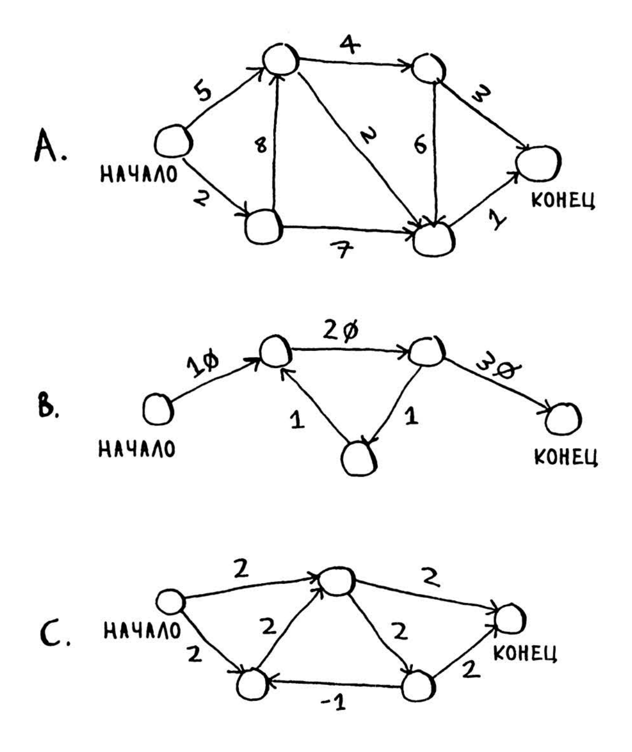
6.5 Какие из следующих графов также являются деревьями?



Ответы: А — дерево; В — не дерево; С — дерево. В последнем примере дерево просто повернуто набок. Деревья составляют подкатегорию графов, поэтому любое дерево является графом, но граф не обязательно является деревом.

Глава 7

7.1 Каков вес кратчайшего пути от начала до конца в каждом из следующих графов?



Ответы: A — 8; B — 60; C — каверзный вопрос (кратчайший путь не существует из-за наличия цикла с отрицательным весом).

Глава 8

8.1 Вы работаете в фирме по производству мебели и поставляете мебель по всей стране. Коробки с мебелью размещаются в грузовике. Все коробки имеют разный размер, и вы стараетесь наиболее эффективно использовать доступное пространство. Как выбрать коробки для того, чтобы загрузка имела максимальную эффективность? Предложите жадную стратегию. Будет ли полученное решение оптимальным?

Ответ: Жадная стратегия заключается в том, чтобы выбрать самую большую коробку, помещающуюся в оставшемся пространстве, и повторять это до тех пор, пока еще можно выбрать хотя бы одну коробку. Нет, такое решение оптимальным не будет.

8.2 Вы едете в Европу, и у вас есть 7 дней на знакомство с достопримечательностями. Вы присваиваете каждой достопримечательности стоимость в баллах (насколько вы хотите ее увидеть) и оцениваете продолжительность поездки. Как обеспечить максимальную стоимость (увидеть все самое важное) во время поездки? Предложите жадную стратегию. Будет ли полученное решение оптимальным?

Ответ: Выбирайте достопримечательность с наибольшей стоимостью в баллах, которую вы успеете посетить в оставшееся время. Остановитесь, когда таких достопримечательностей не останется. Нет, такое решение оптимальным не будет.

Для каждого из приведенных ниже алгоритмов укажите, является ли этот алгоритм жадным или нет.

8.3 Быстрая сортировка.

Ответ: Нет.

8.4 Поиск в ширину.

Ответ: Да.

8.5 Алгоритм Дейкстры.

Ответ: Да.

8.6 Почтальон должен доставить письма в 20 домов. Ему нужно найти кратчайший путь, проходящий через все 20 домов. Является ли эта задача NP-полной?

Ответ: Да.

8.7 Имеется задача поиска максимальной *клики* в множестве людей (кликой называется множество людей, каждый из которых знаком со всеми остальными.) Является ли эта задача NP-полной?

Ответ: Да.

8.8 Вы рисуете карту США, на которой два соседних штата не могут быть окрашены в одинаковый цвет. Требуется найти минимальное количество цветов, при котором любые два соседних штата будут окрашены в разные цвета. Является ли эта задача NP-полной?

Ответ: Да.

Глава 9

9.1 Предположим, к предметам добавился еще один: МР3-плеер. Он весит 1 фунт и стоит $1000. Стоит ли брать его?

Ответ: Да. Вы сможете положить в рюкзак МР3-плеер, iPhone и гитару общей стоимостью $4500.

9.2 Предположим, что вы собираетесь в турпоход. Емкость вашего рюкзака составляет 6 фунтов, и вы можете взять предметы из следующего списка. У каждого предмета имеется стоимость; чем она выше, тем важнее предмет:

- Вода, 3 фунта, 10
- Книга, 1 фунт, 3
- Еда, 2 фунта, 9
- Куртка, 2 фунта, 5
- Камера, 1 фунт, 6

Как выглядит оптимальный набор предметов для похода?

Ответ: Возьмите воду, еду и камеру.

9.3 Нарисуйте и заполните таблицу для вычисления самой длинной общей подстроки между строками *blue* и *clues*.

Ответ:

	C	L	U	E	S
B	0	0	0	0	0
L	0	1	0	0	0
U	0	0	2	0	0
E	0	0	0	3	0

Глава 10

10.1 В примере с Netflix сходство между двумя пользователями оценивалось по формуле расстояния. Но не все пользователи оценивают фильмы одинаково. Допустим, есть два пользователя, Йоги и Пинки, вкусы которых совпадают. Но Йоги ставит 5 баллов любому фильму, который ему понравился, а Пинки более разборчива и ставит «пятерки» только самым лучшим фильмам. Вроде бы вкусы одинаковые, но по метрике расстояния они не являются соседями. Как учесть различия в стратегиях выставления оценок?

Ответ: Можно воспользоваться ***нормализацией***: вы вычисляете среднюю оценку для каждого человека и используете ее для масштабирования оценок. Например, вы определили, что средняя оценка Пинки равна 3, а средняя оценка Йоги – 3,5. Соответственно оценки Пинки немного увеличиваются так, чтобы ее средняя оценка тоже была равна 3,5. После этого оценки можно сравнивать по единой шкале.

10.2 Предположим, Netflix определяет группу «авторитетов». Скажем, Квентин Тарантино и Уэс Андерсон относятся к числу авторитетов Netflix, поэтому их оценки оказывают более сильное влияние, чем оценки рядовых пользователей. Как изменить систему рекомендаций, чтобы она учитывала повышенную ценность оценок авторитетов?

Ответ: При применении алгоритма *k* ближайших соседей можно увеличить вес оценок авторитетов. Предположим, у вас трое соседей: Джо, Дэйв и Уэс Андерсон (авторитет.) Они поставили фильму «Гольф-клуб» оценки 3, 4 и 5 соответственно. Вместо того чтобы вычислять среднее арифметическое их оценок (3 + 4 + 5 / 3 = 4 звезды), вы просто повышаете вес оценки Уэса Андерсона: 3 + 4 + 5 + 5 + 5 / 5 = 4,4 звезды.

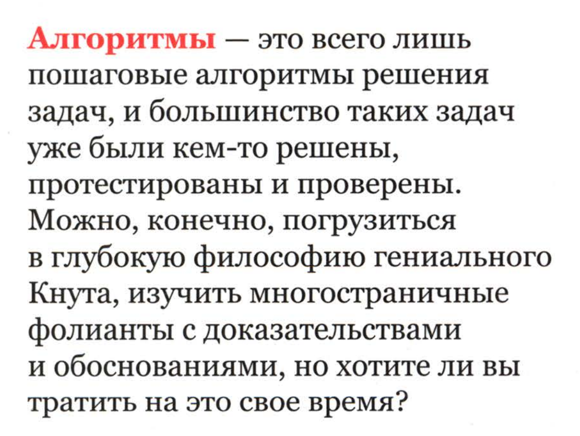
10.3 У сервиса Netflix миллионы пользователей. В приведенном ранее примере рекомендательная система строилась для пяти ближайших соседей. Пять — это слишком мало? Слишком много?

Ответ: Слишком мало. Если ограничиться малым числом соседей, существует высокая вероятность того, что результаты будут искажены. Существует хорошее эмпирическое правило: для *N* пользователей следует рассматривать *sqrt*(*N*) соседей.

Грокаем алгоритмы

Иллюстрированное пособие для программистов и любопытствующих

Адитья Бхаргава



Алгоритмы — это всего лишь пошаговые алгоритмы решения задач, и большинство таких задач уже были кем-то решены, протестированы и проверены. Можно, конечно, погрузиться в глубокую философию гениального Кнута, изучить многостраничные фолианты с доказательствами и обоснованиями, но хотите ли вы тратить на это свое время?

Откройте великолепно иллюстрированную книгу, и вы сразу поймете, что алгоритмы — это просто. А грокать алгоритмы — это веселое и увлекательное занятие.







Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин